MCR-74-314
NAS9-13616

Phase 1
Final
Report

NASA CR-
140363

November 1974

Appendix

Study Approach and Activity Summary

Scheduling Language and Algorithm Development Study

MARTIN MARIETTA

DRL T-890
Line Item 3

MCR-74-314
NAS9-13616

Phase 1
Final
Report

November 1974

Appendix

Study Approach and Activity Summary

SCHEDULING LANGUAGE AND ALGORITHM DEVELOPMENT STUDY

MARTIN MARIETTA CORPORATION
P.O. Box 179
Denver, Colorado 80201

FOREWORD

This is the Phase 1 Final Report of the Scheduling Language and Algorithm Development Study performed by Martin Marietta Corporation, Denver Division, under Contract NAS9-13616. The purpose of this study was to conceive and specify a high-level computer programming language and a program library to be used in writing programs for scheduling complex systems such as the Space Transportation System. This report is presented in three volumes plus an appendix:

Volume I - Study Summary and Overview

Volume II - Use of the Basic Language and Module Library

Volume III - Detailed Functional Specification for the Basic Language and the Module Library

Appendix - Study Approach and Activity Summary

Volume I summarizes the objectives and requirements of the study and discusses the "why" behind the objectives and requirements. Unique results achieved during the study or unique features of the specified language and program library are then described and related to the "why" of the objectives and requirements. Finally, a description of the significance of study results, in terms of expected benefits, is provided.

Volume II summarizes the capabilities of the specified scheduling language and the program module library. It is written with the potential user in mind and, therefore, provides maximum insight on how the capabilities will be helpful in writing scheduling

programs. Simple examples and illustrations are provided in Volume II to assist the potential user in applying the capabilities of his problem.

The detailed functional specifications presented in Volume III are the formal product of Phase 1. These specifications are written as requirements for software implementation of the language and the program modules, and are aimed at a specific audience.

A separate Appendix summarizes the analyses, describes the approach used to identify and specify the capabilities required in the basic language, and presents results of the algorithm and problem modeling analyses used to define specifications for the scheduling module library. The appendix is directed toward the reader who is interested in how the study conclusions and results were reached.

CONTENTS

Figure

Table

1.0 INTRODUCTION

This Appendix to the Phase 1 Final Report of the Scheduling Language and Algorithm Development Study contains three major chapters in addition to this Introduction. Chapter 2.0 describes, in general terms, the approach and organization of the study. Chapter 3.0 documents analyses performed in the first six months of the Phase 1 study by briefly summarizing the material presented in the two volumes of the First Interim Report issued in January 1974. The analyses performed in the second part of the Phase 1 Study to produce the functional specifications for the scheduling language and module library and to perform implementation feasibility, are documented in Chapter 4.0. These analyses were performed in the period between 1 January 1974 and 5 November 1974.

2.0 STUDY APPROACH

The approach to the functional specification of the scheduling language, PLANS (Programming Language for Allocation and Network Scheduling), and the module library used a mix of problem analyses and language design activities. Three major tasks are referred to in this report. Task 1 dealt with the development of the basic language features, and the analysis of implementation options; Task 2 developed methods to describe or model an operational system to be scheduled; and Task 3 identified mathematical and logical strategies for solving scheduling problems. Tasks 2 and 3 effort identified functional requirements for the basic language and appropriate software modules to enhance the capability of the analyst to address realistic problems. In addition, Tasks 2 and 3 served to verify the adequacy and efficiency of the trial language by assessing its functional compatibility with either problem modeling or algorithm applications. Thus, the three tasks worked in an iterative fashion to evolve language-related capabilities that are highly relevant to practical problems. This approach is illustrated conceptually in Fig. A-1.

In the first six months of the study, the general scheduling problem was analyzed from the functional point of view, using a broad range of representative problems. The objective in that time period was to identify and evaluate language features that would satisfy the functional requirements and meet the design goals of (1) usability by a problem analyst and (2) insensitivity to a problem alteration.

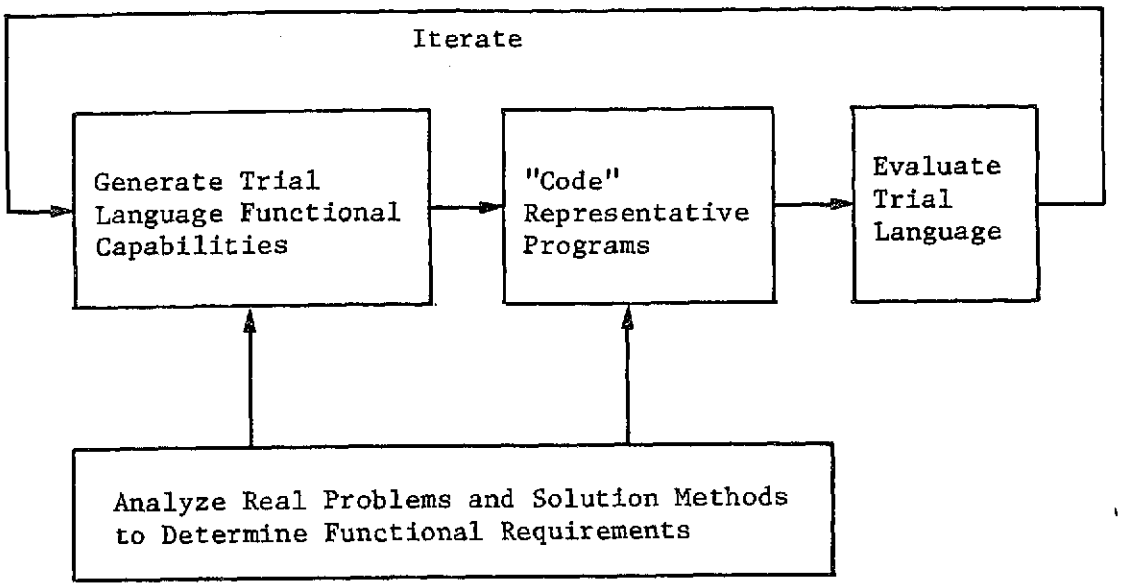


Figure A-1 Iterative Approach to Functional Specification

In the second part of the Phase 1 Study, the Task 1 emphasis was on development of precise syntactic and semantic specifications for PLANS while the Task 2 and Task 3 efforts were directed at functionally specifying a set of library routines that:

1) contained basic functionally-separable logic;
2) were usable in typical scheduling software; and
3) were free from imbedded decisions or assumptions that would restrict the flexibility of their use.

The progress of the Phase 1 study has been guided by milestones for each task. These milestones are shown in Fig. A-2.

Although the study was segmented into discrete subtasks in Fig. A-2, the high degree of integration required to move from conceptual objectives to concrete functional specifications made separation of the total activity into such well-defined subtasks somewhat artificial. For purposes of documentation, in the first Interim Report the subtasks of Fig. A-2 were grouped into major activities; that same format is used here so the reader can better perceive the integrated analyses that have been carried out. Table A-1 lists the major activities that are addressed in this appendix, with reference to the milestone identifier of Fig. A-2.

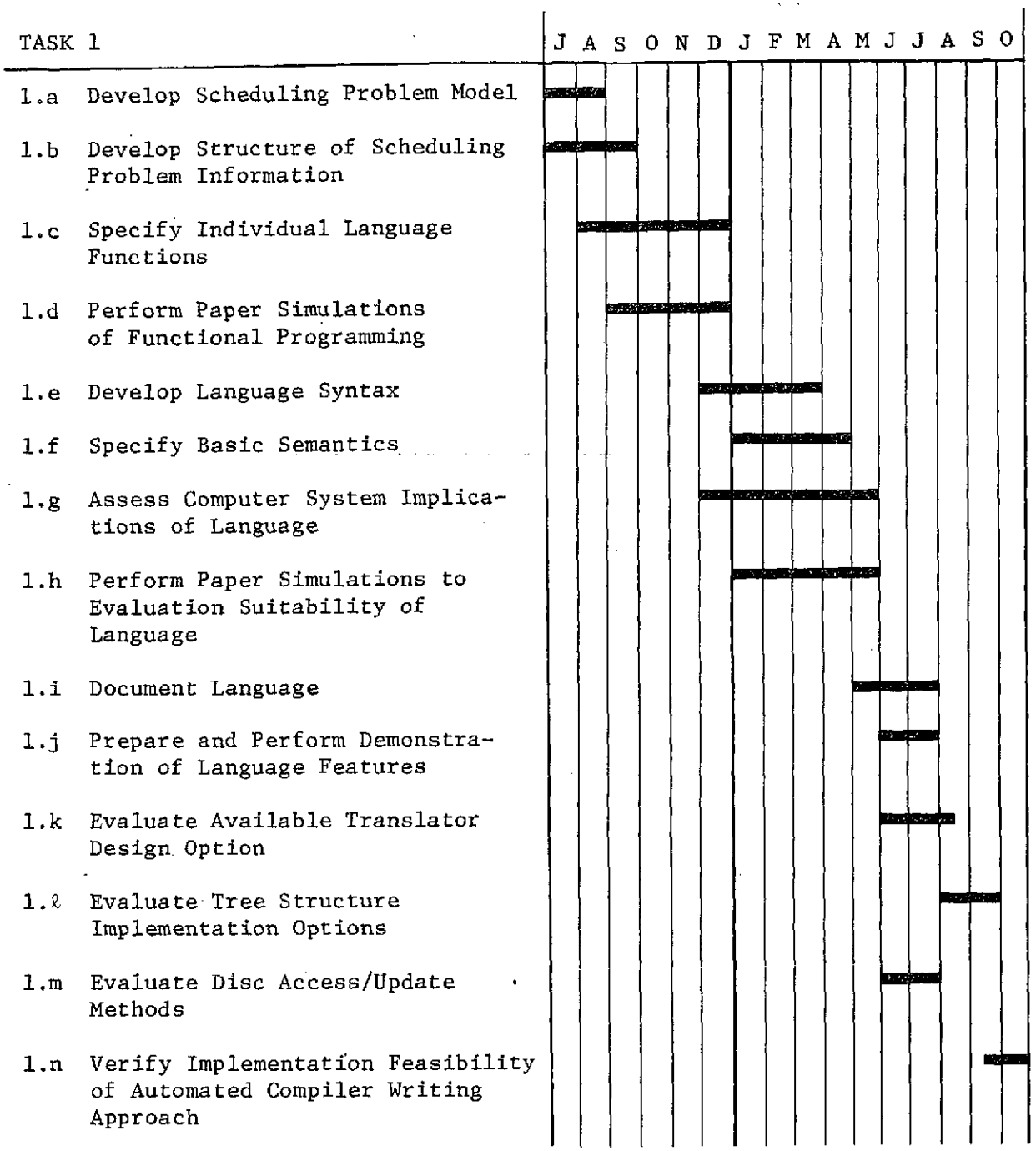


Figure A-2 Milestones

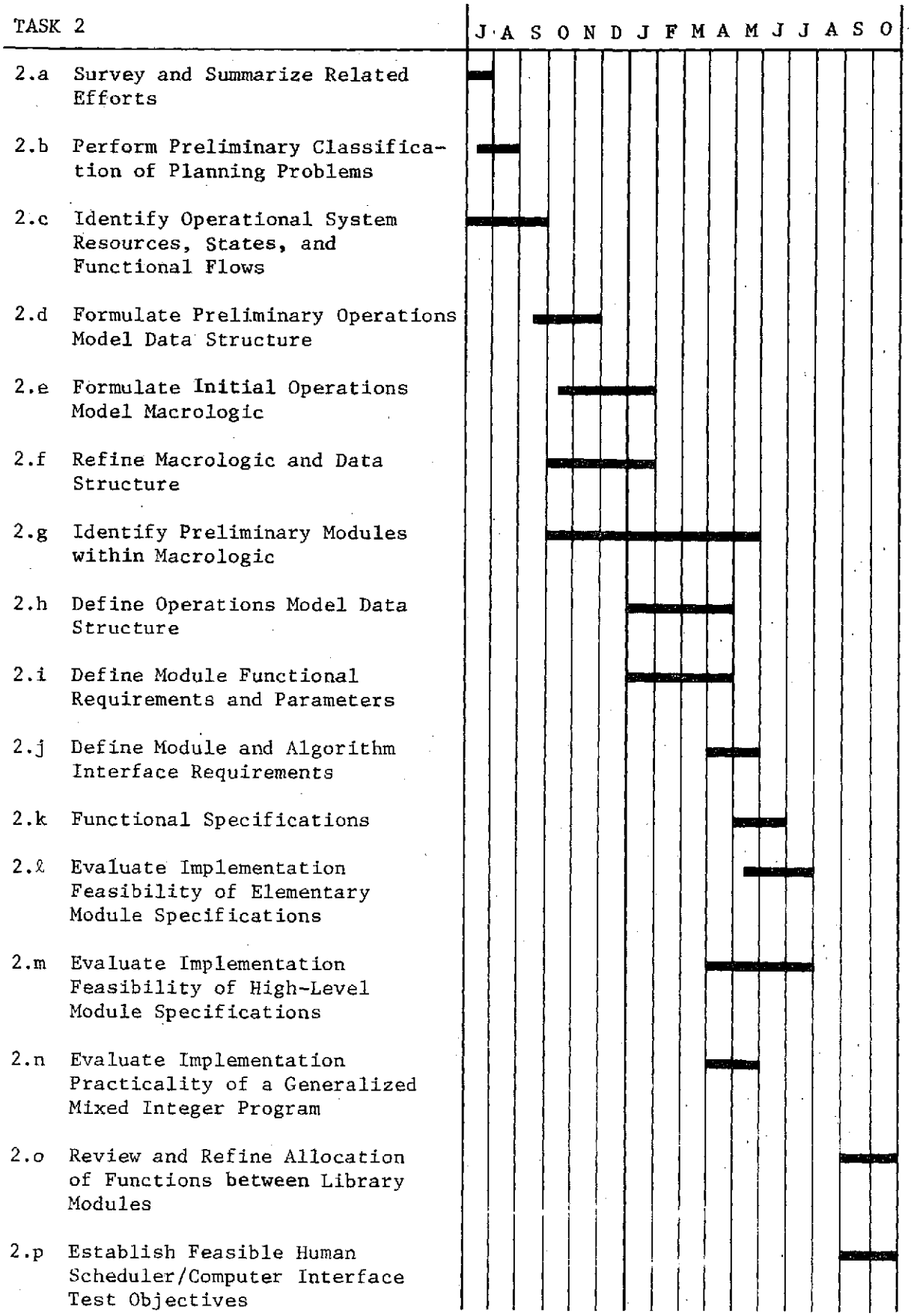


Figure A-2 (cont)

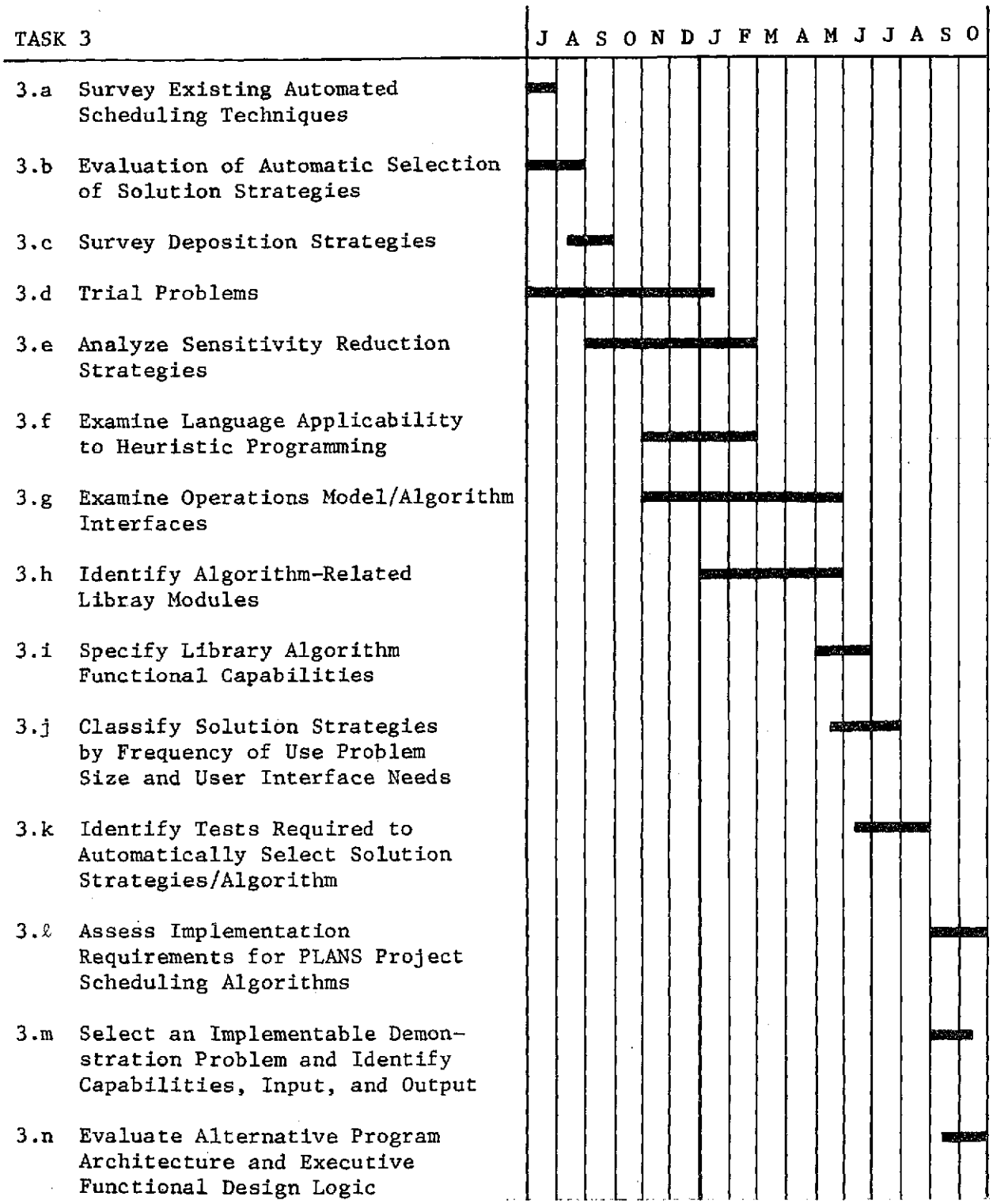


Figure A-2 (concl)

Table A-1 Major Activies and Study Milestone Identifiers

Major Activity	Appendix Section	Milestone Identifier	Milestone Title
Analysis of Structure of the General Scheduling Problem	3.1	1.a	Develop Scheduling Problem Model
		1.b	Develop Structure of Scheduling Problem Information
Formulation of Basic Language Functional Requirements	3.2	1.c	Specify Individual Language Functions
Synthetic Programming for Trial Language Evaluation	3.3	1.d	Perform Paper Simulations of Functional Programming
Assessment of Scheduling Operations Model Requirements	3.4	2.a	Survey and Summarize Related Efforts
		2.b	Perform Preliminary Classification of Scheduling Problems
		2.c	Identify Operational System Resources, Status, and Functional Flows
Synthesis and Functional Evaluation of Operations Model	3.5	2.d	Formulate Preliminary Operations Model Data Structure
		2.e	Formulate Initial Operations Model Macrologic
		2.f	Test and Refine Macrologic and Data Structure
		3.g	Examine Operations Model/Algorithm Interfaces
		2.j	Define Module and Algorithm Interface Requirements
Analysis of Solution Techniques Applicable to Scheduling Problems	3.6	3.a	Survey Existing Automated Scheduling Techniques
		3.c	Survey Decomposition Strategies
		3.f	Analyze Sensitivity Reduction Strategies
		3.g	Examine Language Applicability to Heuristic Programming
Identification of Language Requirements via Solution of Trial Problems	3.7	3.b	Evaluation of Automatic Selection of Scheduling Strategies
		3.d	Trial Problems
Formulation of Prelimary List of Library Modules	3.8	2.g	Identify Preliminary Operations Model Modules within Macrologic
		3.h	Identify Algorithm-Related Library Modules
Preliminary Assessment of Language Translation Options	3.9	1.g	Assess Computer System Implications of the Language
Development of Mechanism for Syntactic/Semantic Specification	4.1	1.e	Develop Language Syntax
		1.f	Specify Basic Semantics
		1.i	Document Language
Evaluation of Language Suitability for Applications	4.2	1.h	Perform Paper Simulations to Evaluate Suitability of Language

Table A-1 (concl)

Major Activity	Appendix Section	Milestone Identifier	Milestone Title
Evaluation of Language Implementation Feasibility	4.3	1.g	Assess Computer System Implications of Language
		1.k	Evaluate Available Translator Design Option
		1.ℓ	Evaluate Tree Structure Implementation Options
		1.m	Evaluate Disc Access/Update Methods
		1.n	Verify Implementation Feasibility of Automated Compiler Writing System
Development of Contents and Specifications for Library Modules	4.4	2.i	Define Module Functional Requirements and Parameters
		2.k	Functional Specifications
		3.f	Examine Language Applicability to Heuristic Programming
		3.g	Examine Operations Model/Algorithm Interfaces
		3.h	Identify Algorithm-Related Library Modules
		3.i	Specify Library Algorithm Functional Capabilities
		2.o	Review and Refine Allocation of Functions between Library Modules
Development of Standard Data Structures	4.5	2.d	Formulate Preliminary Operations Model Data Structure
		2.h	Define Operations Model Data Structure
		2.j	Define Module and Algorithm Interface Requirements
Assessment of Implementation Feasibility of Specified Modules	4.6	2.ℓ	Evaluate Implementation Feasibility of Elementary Module Specifications
		2.m	Evaluate Implementation Feasibility of High-Level Module Specifications
		2.n	Evaluate Implementation Practicality of a Generalized Mixed Integer Program
		3.ℓ	Assess Implementation Requirements for PLANS Project Scheduling Algorithms
		3.m	Select Implementable Demonstration Problem and Identify Capabilities, Input, Output
		3.n	Evaluate Alternative Program Architecture and Executive Functional Design Logic
Assessment of Methods for Automated Algorithm Application	4.7	3.j	Classifify Solution Strategies by Frequency of Use Problem Size and User Interface Needs
		3.k	Identify Tests Required to Automatically Select Solution Strategies/Algorithm
		2.p	Establish Feasible Human Scheduler/Computer Interface Test Objectives

3.0 SUMMARY OF MAJOR ACTIVITIES IN THE FIRST PART OF STUDY PHASE 1

The major activities of the first six months of the study are summarized in this chapter. It contains a brief description of the conclusions reached and the analyses leading to those conclusions. Numerous references are made to Volume II of the First Interim Report, where detailed documentation is presented.

3.1 ANALYSIS OF STRUCTURE OF THE GENERAL SCHEDULING PROBLEM

The scheduling language study was initiated by defining a basic structure within which language functional requirements could be developed. It was recognized that assignment of resources for intervals of time is fundamental to the concept of a schedule. Therefore, a schedule unit was defined as a collection of the assignments for specific resources. It naturally follows that a schedule is a collection of schedule units. A simple illustration of a schedule is given in Fig. A-3. Note that a schedule unit contains assignment intervals that may be different for each resource in the schedule unit. Precise definitions of the terms "Schedule Unit," "Schedule," and "Scheduling Problem" are given in Section 3.1 of Volume II.

The analysis of the structure of scheduling problems continued with examining the information required for scheduling. The objective was to find how this information was most naturally organized so appropriate data structures for a scheduling language could be identified. The analysis revealed that all information was hierachically related. Even though a single universal format

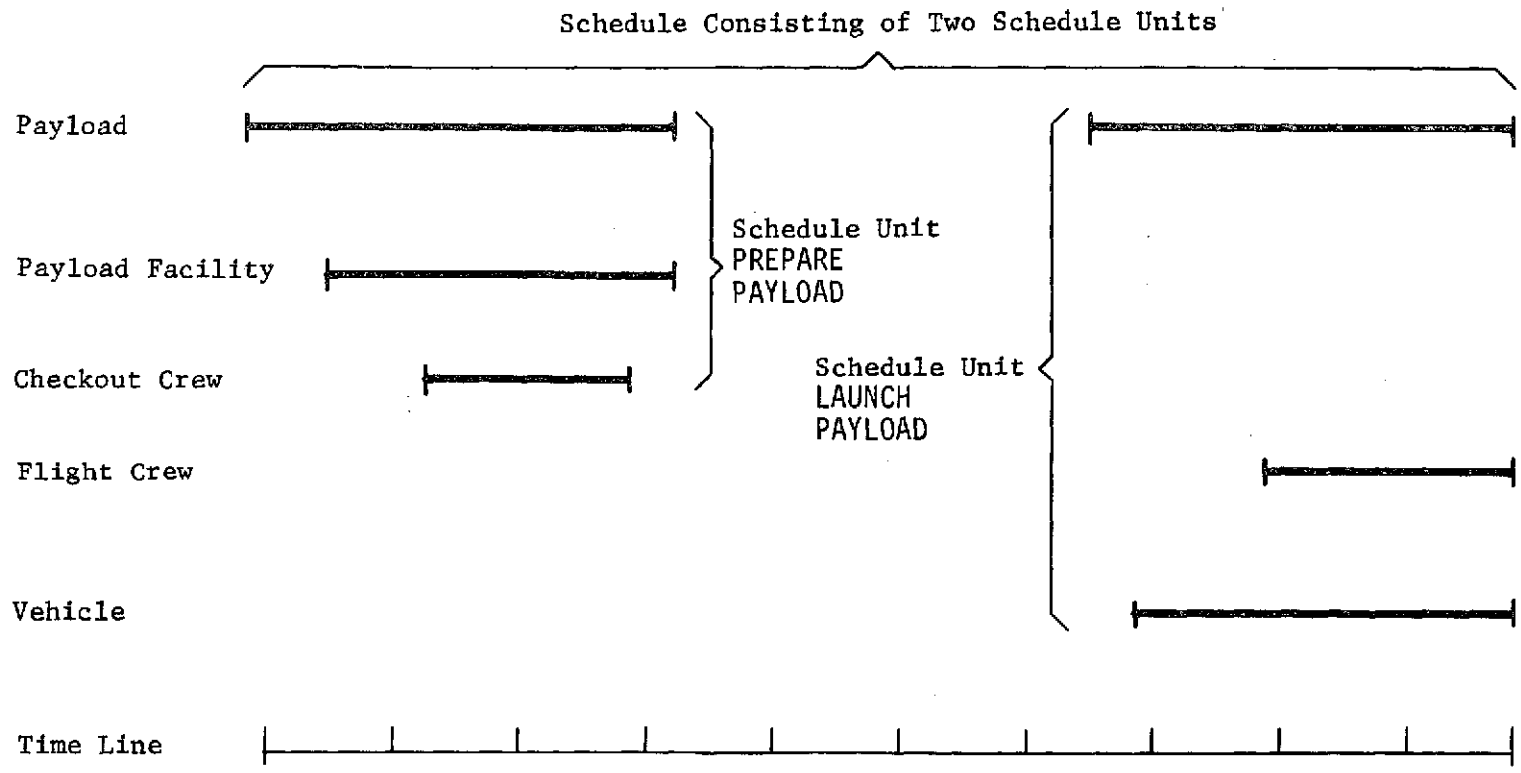


Fig. A-3 Illustration of the Structure of a Schedule

for scheduling problem information is not feasible, the conclusion was that a hierarchical structure appears appropriate for all problems. Thus, a basic data type had been identified for PLANS.

3.2. FORMULATION OF BASIC LANGUAGE FUNCTIONAL REQUIREMENTS

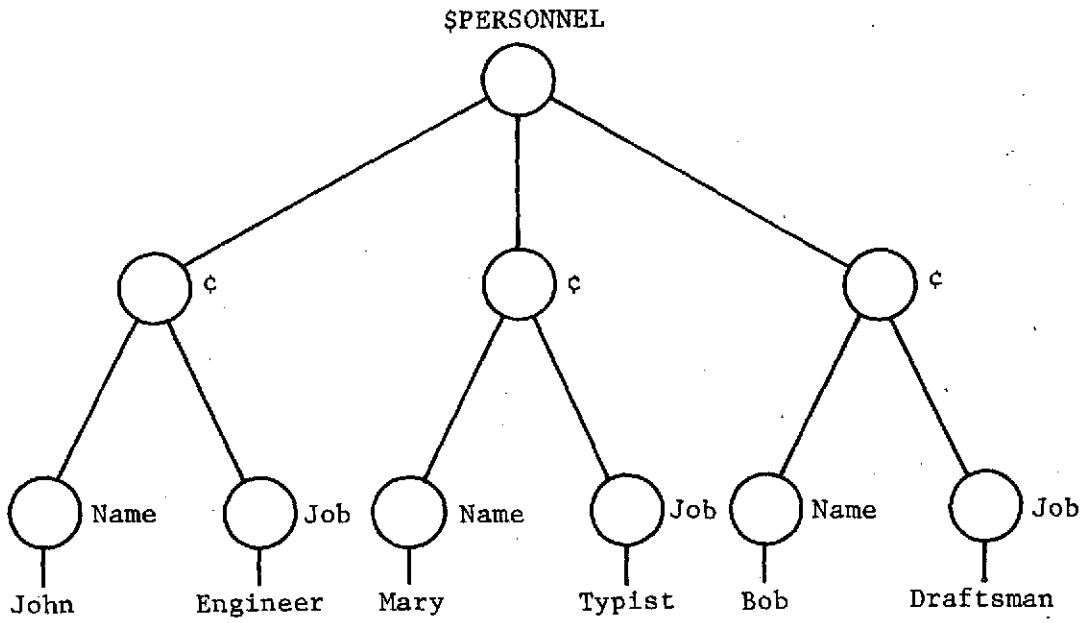
Analysis of the structure of general scheduling problems was followed by preliminary identification of the elementary functional capabilities that a scheduling language should possess.

To appreciate the approach to defining functional requirements, it is necessary to understand a fundamental distinction between the *basic capabilities* of a language and the capabilities that the language lends to the programmer. For example, it is possible to integrate a function using FORTRAN, but integration is not, in any sense, a basic language capability. The basic FORTRAN capabilities of array manipulation, algebraic operations, and iteration allow the programmer to perform integration. Only if FORTRAN included something functionally equivalent to the command INTEGRATE FUNCTION X, would it be appropriate to say that integration is a *basic capability* of FORTRAN.

The principal task associated with design of PLANS during the first six months of this study was to extract a list of underlying elementary operations that must be performed by single language statements (or even by part of a statement).

The basic language characteristics identified for PLANS are described in detail in Section 3.3, Volume II of the First Interim Report. The basic data type identified is the hierarchical set structure. Although intervals have a special place in scheduling problems and were originally identified as a second data type, subsequent analyses showed that intervals could be handled without difficulty within the tree structure and by specifying a small number of interval subroutines. Thus, specification of the hierarchical structure as the only required data type provides logical simplicity and considerably greater economy of implementation than would result from a variety of data types. An illustration of a hierarchical set structure is given in Fig. A-4. PLANS must have the capability to generate and to alter hierarchical structures and to access the contents of the structure either by key word (label) or by ordinal position (index). It is significant that, although the need for hierarchical data became evident early in the study, subsequent analyses have continually reinforced its relevance and importance in achieving language power.

Functional capabilities for PLANS identified in this activity include (1) algebraic operations, (2) input/output operations, (3) transfer of control statements, (4) conditional statements, (5) function and/or subroutine capabilities, and (6) iteration statements.



Note: The symbol $ is used as a prefix to identify the label of a data tree root node. Reference to $PERSONNEL within a PLANS program statement refers to the data tree (hierachical data set) root node label and all data contained within the tree, thus

$PERSONNEL

¢ Name - John
Job - Engineer

¢ Name - Mary
Job - Typist

¢ Name - Bob
Job - Draftsman

In a loose definition, $PERSONNEL may be referred to as the data set for PERSONNEL.

The symbol ¢ has been adopted to indicate a null label.

Fig. A-4 Hierarchical Set Structure

An operation that frequently occurs in scheduling is the generation of combinations or permutations of a given set. Therefore, special PLANS iteration capabilities have been identified that will generate, one at a time, all the combinations or permutations of a set taken K at a time.

Because a design goal for PLANS is a programming capability that is as independent as possible of application-specific information, a requirement for indirect referencing was identified. Two kinds of indirect referencing are required. The first is indirect reference to a set, or within that set. An example of this capability can be seen in the following hypothetical language statements:

```
        DO 15  J = 1, N
15      TOTALWT = TOTALWT + $RESOURCE.#($COMPONENT(J).NAME).WEIGHT
```

The symbol # is used here to indicate an indirect reference. If $COMPONENT has the structure

```
$COMPONENT
1
  NAME - ORBITER 7
2
  NAME - PAYLOAD 35
3
  NAME - SRM 12
```

then the iteration loop above sums the weight of Orbiter 7, Payload 35, and SRM 12. The weight information is found in the $RESOURCE set. Note that if the weight of Orbiter 7, Payload 35, and Crewmen 12 were desired, only the $COMPONENT data would have to be changed, the code would remain the same because the labels ORBITER 7, PAYLOAD 35, SRM 12, never appeared in the program logic. This illustrates the reason for the first type of indirect reference capability.

The second type of indirect referencing deals with module names. Consider, for example, the statement:

```
CALL #($RESOURCE(J).RECYCLE)
```

The name of a subroutine that calculates the value of the recycle time of $RESOURCE(J) could be included as data as follows:

```
$RESOURCE
1
  RECYCLE - ORBCYC
2
  RECYCLE - PADCYC
.
.
.
```

As the CALL statement is encountered with different values of J, different recycle modules are called. This coding can be written both concisely, and also independently of orbiters, launch pads, etc.

Another functional capability identified is set ordering. A single "order" statement can arrange the elements of a set in order (ascending or descending) according to a list of characteristics of that set. For example, the statement:

```
ORDER $PAYLOADS ON WINDOW.START,WINDOW.END
```

would create a payload ordering in which earlier window openings preceded later openings. Two or more windows with equal opening times would be ordered so those with earlier closing times preceded those with later closing times.

3 SYNTHETIC PROGRAMMING FOR TRIAL LANGUAGE EVALUATION

By the end of the second study month, the initial list of elementary language operations had been identified and it included many of the features described in the preceding section. It was desirable to test the functional validity of the basic language operations for scheduling problems, so a trial FORTRAN-like syntax was adopted, although it was considered subject to later replacement or revision. The syntax made it possible to code complete language statements and programs in this trial language, sometimes called Trial PLANS.

Synthetic programming was then performed using the trial language to evaluate basic language functional capabilities in realistic program applications. The programming was "synthetic" in the sense that no means existed for translation to machine code for actual program execution. Three types of programs, described in subsequent paragraphs, were coded and yielded further insights and requirements for language design.

Coding with Trial PLANS was used to synthesize a number of small routines of general utility in solving scheduling problems and also to program several larger main programs. The main programs coded in Trial PLANS included an algorithm selection program that performs tests on the scheduling problem structure to select candidate solutions.

Another program coded in Trial PLANS was a critical path method (CPM) program. The program finds the earliest and latest start times for a set of jobs that must be completed in a sequence network, determines the job that cannot be delayed without extending the duration of the entire project (i.e., jobs on the critical path), and the slack times for all other jobs. This program provided an excellent example of the ease with which ordering and set manipulations can be performed with Trial PLANS.

Synthetic programming was also used to code the basic logic of the NASA JSC-MPAD Operations Simulation and Resource Scheduling program (OSARS). This exercise demonstrated that the basic language capabilities identified did make flexible programming possible, and that the basic capabilities alone provided a level of coding statements approximately equivalent to basic logic elements in a functional flow diagram. The number of PLANS statements in the PLANS-OSARS program is approximately one-tenth of that required by a FORTRAN version.

3.4 ASSESSMENT OF SCHEDULING OPERATIONS MODEL REQUIREMENTS

Scheduling involves making decisions about alternatives in the operations of a system. Because the task was to develop both the functional specifications for a scheduling language and a library of program modules, it was imperative to specify a framework within which the operations of the system to be scheduled could be described. Such a framework must be completely compatible with the basic language. Furthermore, many of the higher-level modules in the module library must be designed to

use a standard descriptive framework so they can be easily and consistently applied within the logic of a calling program. A number of technical disciplines already deal with the description and response of dynamic systems. Because scheduling is itself not a new problem, it was necessary to carefully review existing technology before this study specified a general scheduling operations model (or, briefly, the Operations Model) for the scheduling language.

The complexity of building a generic operations model required a top-down approach to guarantee that structures specified were sufficiently general to preclude making decisions that must subsequently be revised. The approach began with the concept that system resources exist with various descriptors that are transformed or altered by system processes. This fundamental concept is illustrated in Fig. A-5. Noting that a process must occur over a time interval, it is recognized that the process is, in fact, the entity that associates resources together in a schedule unit. A given process has a set of required resources, and those resources must have appropriate descriptors. The execution of a process, then, is functionally equivalent to specifying assignment intervals for the processes' required resources.

Continued analysis led to identification of the operations sequence as a mechanism for describing how various processes are related to each other in time (temporally) and as predecessors and successors. For example, an operations sequence might contain the information that process A must be concluded before process B starts, or that process B must start after process C starts.

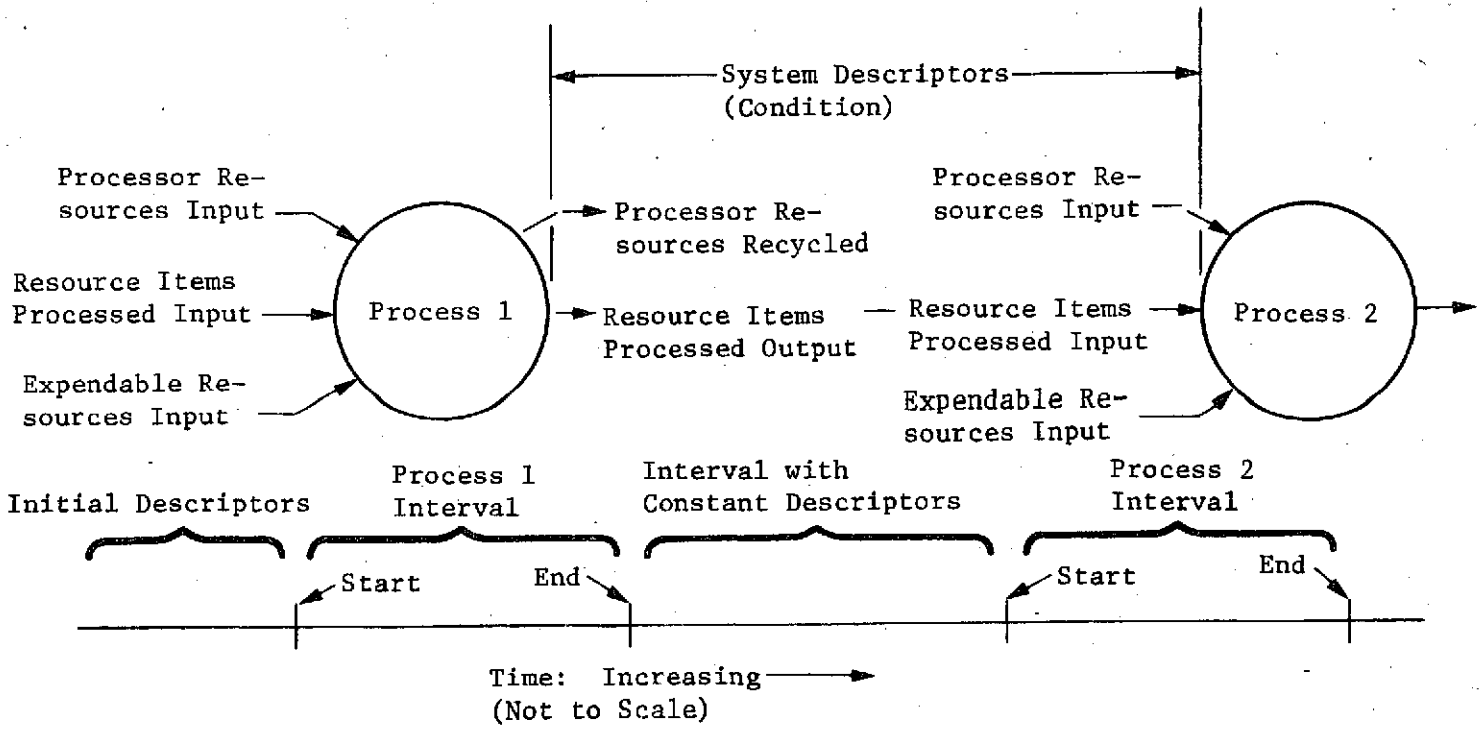


Fig. A-5 Operations Model Fundamental Concepts

After expansion of the fundamental system operations model components, i.e., the resources, processes, and operational sequences, an evaluation of the feasibility of integrating these components into a model of a system that can be scheduled could then be undertaken.

1.5 SYNTHESIS AND FUNCTIONAL EVALUATION OF THE OPERATIONS MODEL

The compatibility of the fundamental modeling concepts with the functional capabilities of the PLANS language is, of course, essential. Therefore, the structure for describing the system to be scheduled (i.e., resources, processes, operations sequences), with the hierarchical data structure identified as a language feature, was examined. It was determined that the information could be organized into three tree structures called $OPSEQUENCE, $PROCESS and $RESOURCE. Examples of these structures for a model of Shuttle operations are shown in Table A-2.

Table A-2 illustrates one of the primary features of the scheduling operations model data structure--the use of labels for data entries. Although use of labels within the data structure adds volume, their functional usefulness for accessing data is more than sufficient justification. Also, the labels make the data "readable," thus eliminating the need for tedious references to a user's manual to determine formats. A brief examination of Table A-2 will enable readers to develop a basic understanding of how models can be specified in the hierarchical form. The fact that no detailed explanation of the model data structure is necessary conveys the point about readability. The logical

Table A-2 Examples of Operations Model Data Structures

```
$RESOURCE
   SRB
      SRB
         QUANTITY - 18
         CLASS - POOL

   VAB HIGH BAY
      HIGH BAY NO. 1
         QUANTITY - 1
         LOCATION - BAY 1
         CLASS - SPECIFIC

      HIGH BAY NO. 2
         QUANTITY -1
         LOCATION - BAY 2
         CLASS - SPECIFIC

   PERSONNEL
      SRB/EXT. TANK CREW
         QUANTITY - 55
         QUALIFICATIONS - ASSEMBLE SRBS
                        - PREPARE EXT. TANKS
                        - MATE TANK AND SRB
                        - REFURBISH LUT
         CLASS - POOL

      LAUNCH CREW
         QUANTITY - 75
         QUALIFICATIONS - SERVICE SHUTTLE
                        - LAUNCH OPS
                        - REFURBISH PAD
         CLASS - POOL

      ORBITER/PAYLOAD CREW
         QUANTITY - 45
         QUALIFICATIONS - PERFORM PAYLOAD OPS
                        - PREPARE ORBITER
                        - RECYCLE ORBITER
                        - MATE ORBITER AND TANK
         CLASS - POOL
         ASSIGNMENT -
      MISSION CONTROL
         QUANTITY - 65
         QUALIFICATIONS - ON-ORBIT OPS
                        - DEORBIT AND LAND
         CLASS - POOL

      CREW OPS
         QUANTITY - 75
         QUALIFICATIONS - CREW TRAINING
                        - MISSION BRIEFING
                        - FLIGHT CREW PREP
                        - DEBRIEFING
         CLASS - POOL

   LAUNCH UMBILICAL TOWER
      LUT
         QUANTITY - 3
         CLASS - SPECIFIC

   EXT. TANK
      EXT. TANK
         QUANTITY - 7
         CLASS - POOL

   ORBITER
      ORBITER
         QUANTITY -3
         CLASS - SPECIFIC

   LAUNCH PAD
      LAUNCH PAD NO. 1
         QUANTITY - 1
         LOCATION - PAD 1
         CLASS - SPECIFIC

      LAUNCH PAD NO. 2
         QUANTITY - 1
         LOCATION - PAD 2
         CLASS - SPECIFIC

   CREW
      CREW
         QUANTITY -15
         CLASS - POOL

   PAYLOAD
      PAYLOAD
         QUANTITY - 163
```

```
$PROCESS
   RECYCLE SRB
      DURATION - 11
      REQUIRED RESOURCES
         SRB
            SRB
               ¢
                  INTERVAL
                     START - 0
                     END - 11
                  DESCRIPTORS
                     ¢
                        INITIAL
                           QUANTITY - 2
                           STATUS - TO BE RECOVERED
                        FINAL
                           QUANTITY - 2
                           STATUS - TO BE ASSEMBLED

   ASSEMBLE SRB PAIR
      DURATION - 0.4
      REQUIRED RESOURCES
         SRB
            SRB
               ¢
                  INTERVAL
                     START - 0
                     END - 47
                  DESCRIPTORS
                     ¢
                        INITIAL
                           QUANTITY - 2
                           STATUS - TO BE ASSEMBLED
                        FINAL
                           STATUS - TO BE MATED
         VAB HIGH BAY
            VAB
               ¢
                  INTERVAL
                     START - 0
                     END - 47
                  DESCRIPTORS
                     ¢
                        INITIAL
                           STATUS - AVAILABLE
                        FINAL
                           status - IN USE
         PERSONNEL
            SRB/EXT TANK CREW
               ¢
                  INTERVAL
                     START - 0
                     END - 47
                  DESCRIPTORS
                     ¢
                        INITIAL
                           QUANTITY - 30
                           QUALIFICATIONS - ASSEMBLE SRBS
         LAUNCH UMBILICAL TOWER
            LUT
               ¢
                  INTERVAL
                     START - 0
                     END - 47
                  DESCRIPTORS
                     ¢
                        INITIAL
                           QUANTITY - 1
                           STATUS - AVAILABLE
                        FINAL
                           STATUS - IN USE
         RESOURCES GENERATED
            SRB PAIR
               SRB PAIR
                  ¢
                     DESCRIPTORS
                        ¢
                           FINAL
                              QUANTITY - 1
         RESOURCES DELETED
            SRB
               SRB
                  ¢
                     DESCRIPTORS
                        ¢
                           INITIAL
                              QUANTITY - 2
                           FINAL
                              QUANTITY - 0
```

```
$OPSEQ
   SHUTTLE SYSTEM MISSION FLOW
      ASSEMBLE SRB PAIR
         TYPE - PROCESS
      PREPARE EXT. TANK
         TYPE - PROCESS
      MATE EXT. TANK TO SRB
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               ASSEMBLE SRB PAIR
               PREPARE EXT. TANK
      MATE ORBITER TO EXT. TANK
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               MATE EXT. TANK TO SRB
               PREPARE ORBITER FOR LAUNCH
      SERVICE SHUTTLE FOR LAUNCH
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               MATE ORBITER TO EXT. TANK & SRB
      LAUNCH PHASE OPERATIONS
         TYPE - PROCESS
         TEMPORAL RELATIONS
            GENERAL
               ¢
                  START
                  EQUAL TO
                  END
                  SERVICE SHUTTLE FOR LAUNCH
            PREDECESSOR
               PREPARE CREW FOR FLIGHT
      PREPARE CREW FOR FLIGHT
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               PERFORM MISSION BRIEFING
            GENERAL
               ¢
                  END
                  EQUAL TO
                  END
                  SERVICE SHUTTLE FOR LAUNCH
      REFURBISH LAUNCH PAD
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               LAUNCH PHASE OPERATIONS
      RECYCLE SRB
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               LAUNCH PHASE OPERATIONS
      PERFORM ON-ORBIT OPERATIONS
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               LAUNCH PHASE OPERATIONS
      DEORBIT, REENTRY AND LAND
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               DEORBIT, REENTRY AND LAND
      RECYCLE ORBITER
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               DEORBIT, REENTRY AND LAND
      PERFORM CREW TRAINING OPS
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               PERFORM PAYLOAD OPS
      PERFORM MISSION BRIEFING
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               PERFORM CREW TRAINING OPS
      PERFORM PAYLOAD OPS
         TYPE - PROCESS
      PREPARE ORBITER FOR LAUNCH
         TYPE - PROCESS
         TEMPORAL RELATIONS
            PREDECESSOR
               PERFORM PAYLOAD OPS
```

transparency of the scheduling operations model data structure is essential to ease of program application and adaptation, and eliminates the need for a high level of specialized knowledge to develop a scheduling program.

Recognizing that details of the arrangement of information within the Operations Model data structure depended on how the information was used in a scheduling problem solution, macrologic was developed that described how the Operations Model and the algorithm would function together to solve a scheduling problem. For logical simplicity, the Operations Model was defined as (1) the Operations Model data structure and (2) those functions required to synthesize a schedule that do not involve process alternative, resource allocation, or event timing decisions. Model capabilities in this category include updating the resource assignments, evaluating resource availability, computing values of any special parameters needed by an algorithm to make a decision, etc. With that conceptual distinction, the roles of the model and the algorithm can be interpreted in terms of a dialogue; the algorithm asks for problem-oriented information about the system and its operations on which to base a scheduling decision, and the model supplies the information.

A typical example of the macrologic of the operations model and a time-progressive heuristic algorithm is shown in Fig. A-6. The figure also contains annotations to interpret the macrologic in terms of the OSARS program of NASA-MPAD currently used as a prototype program for building flight schedules.

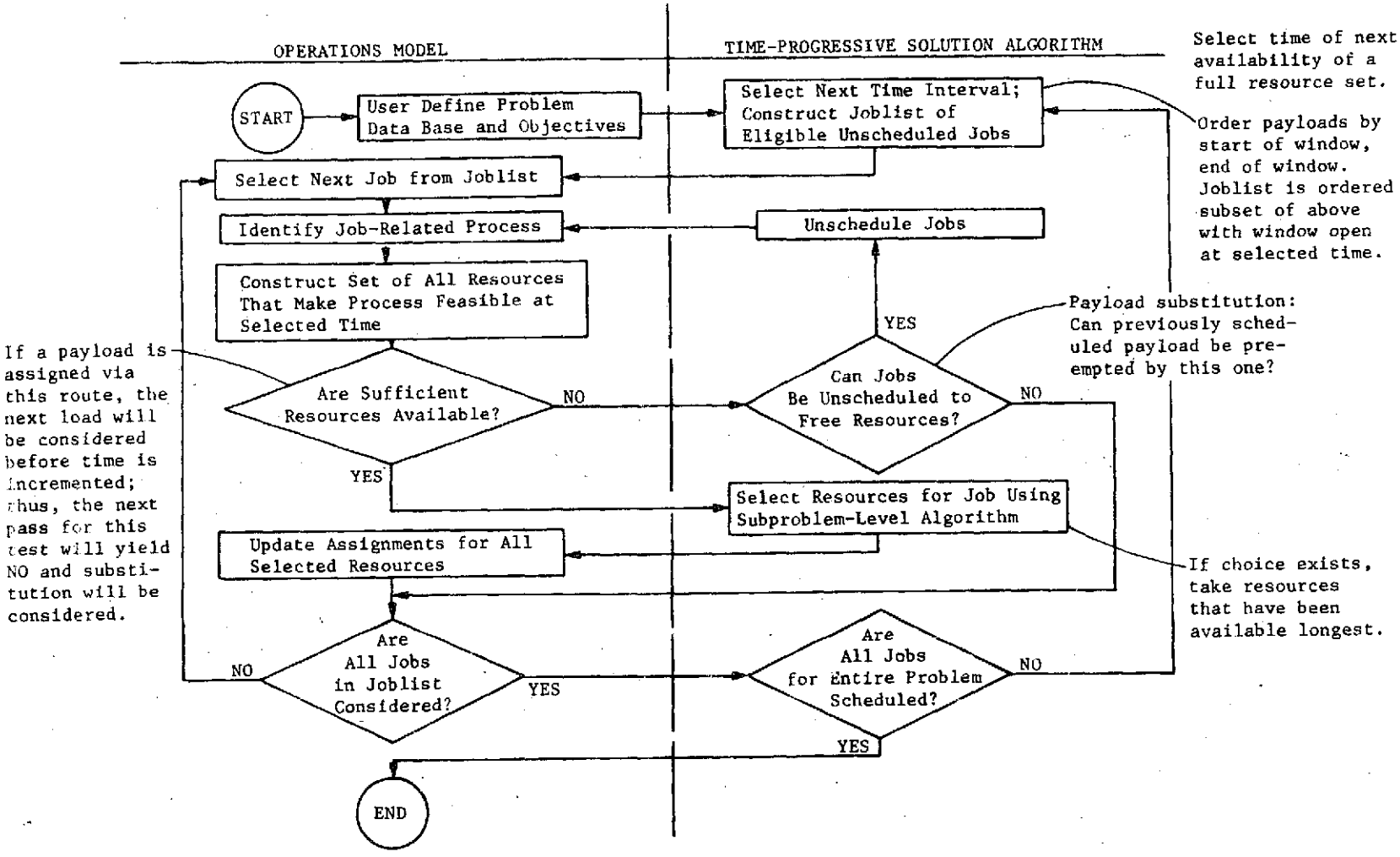


Fig. A-6 Operations Model/Solution Algorithm Interface Example with OSARS Annotations

3.6 ANALYSIS OF SOLUTION STRATEGIES APPLICABLE TO SCHEDULING PROBLEMS

The analysis described here deals with classification of algorithms appropriate for scheduling problems, examination of techniques for decomposing large problems into computationally practical subproblems, and analysis of heuristic methods for solving complex problems.

To design a language applicable to a wide variety of scheduling problems, it is necessary to study a very large number of algorithms. This is accomplished most efficiently within the framework of some logical classification scheme. In the early weeks of the study, such a scheme was developed based on the characteristics of the problems to which certain algorithms applied. Thus, problems were classified from a solution strategy viewpoint. This classification is summarized in Table A-3.

Table A-3 Summary of Algorithm Classification

PROBLEM	ALGORITHM CLASS
Low-Dimensional Simple Scheduling	General-Purpose Mathematical Programming (ILPs, Dynamic Programming)
Medium-Dimensional Specialized Scheduling	Special-Purpose Mathematical Programming (Marshal Fisher, etc)
High-Dimensional Complex Scheduling	Heuristic Algorithms (Wiest, Kelly, etc)
Set Covering (Payloads)	Enumeration (Total, Bounded, Implicit)

The strategy of decomposition is fundamental to the application of many algorithms (especially mathematical programming) to scheduling problems of realistic size and complexity. Decomposition consists of any mathematical or logical technique for working the entire problem by solving smaller and simpler related problems. The relationship of the subproblems is manipulated by a so-called master algorithm that serves to coordinate the subproblems in such a way that optimality of the entire problem is guaranteed.

Several sources exist for methods of problem decomposition. Methods analyzed in this study are summarized in Table A-4. An explanation of each of these techniques appears in Vol II of the First Interim Report.

Table A-4 Summary of Decomposition Strategies

STRATEGY	REFERENCE
Restricted Master, Column Generation	Dantzig and Wolfe (1960)
Dual Minimax	Everett (1963)
Right-Hand Side Allocation	Silverman (1968)
Extended Generalized Upper Bonding	Kaul (1965)
Benders' Decomposition	Benders (1962)
Rosen's Partition	Rosen (1963)

To investigate the problem of algorithm selection that always faces the analyst with a scheduling problem, a logical network (Fig. A-7) has been developed that gives the appropriate sequence of decisions that must be made about problem structure to reach an appropriate algorithm selection.

Fig. A-7 Problem Characteristics Amenable to Mathematical Programming

3.7 IDENTIFICATION OF FUNCTIONAL REQUIREMENTS VIA SOLUTION OF TRIAL PROBLEMS

The approach used to identify appropriate features for PLANS has placed heavy emphasis on analysis of real problems. Ability to determine appropriate functional capabilities requires a thorough understanding of methods for solving scheduling problems that have successfully provided computational results. The choice of solution techniques must be made not only on the basis of problem structure, but on the basis of computational practicality and experience with the details of computational pitfalls and complications. A truly practical collection of library modules for the language must include capabilities to perform special functions and adjustments, and modifications that are almost always necessary to accelerate or improve the computational results. These rather subtle procedures can only be discovered by solving problems. Therefore, in the first six months of the study, a variety of specific scheduling problems were defined. There problem characteristics and structure were thoroughly analyzed, and one or more solution strategies defined for each of the trial problems summarized in Table A-5. The strategies were computerized either by writing FORTRAN programs or using existing programs.

Detailed discussion of trial problem analysis is contained in the First Interim Report.

Table A-5 Summary of Trial Problems

ID	Problem Name	Solution Strategy	Algorithms Employed
1	Activity Scheduling Problem	Decomposition 0-1 Linear Program at Master Level, Enumeration at Subproblem-Level	Generalized Linear Program
2	Multi-Item Scheduling Problem	Decomposition 0-1 Linear Program at Master Level Dynamic Programming at Subproblem-Level	Generalized Upper Bounding Dual Minimax with Steepest Ascent Dynamic Programming
3	Tire-Facility Problem	Two-Level Heuristic Master Level Handles Precedence and Resource Constraints. Subproblem-Level Decides Substitutability	Time Transcendent (Minimum Slack) Algorithm at Master-Level, Minimum Utility Rule at Subproblem-Level
4	Problem of Prittsker, Watters, and Wolfe	One-Level Heuristic (Minimum Slack)	Time Transcendent Heuristic (Minimum Slack)
5	Flowshop and General Combinations Problem	Bounded Enumeration Using Partial Schedules	Ignall and Schrage's Bounded Enumeration Algorithm
6	Set Covering Problem	Total Enumeration for Tractible Numbers of Combinations	Enumeration Tree Tailored to Payload Set Covering Using Domination to Prune Branches
7	Resource Leveling Problem	Solve Minimum Time, Restricted Resource Problems (Possibly a Sequence of Problems Varying Resource Poor Levels)	Weist's Time Progressive Multiresource-Level Heuristic Algorithm

Several potential library modules were identified as a result of the pursuit of specific problems from conceptual definition to numerical results. However, emphasis in the first six months was focused on available experience with computational techniques on the general functional aspects of specific types of scheduling

problems. Because of these efforts, it was possible to structure the findings into a set of discrete algorithm-related modules that are relevant to the range of problems for which PLANS is being designed, and which are useful to the analyst who is confronted with computational subtleties.

3.8 FORMULATION OF PRELIMINARY LIST OF LIBRARY MODULES

Activities in the first six months were designed to establish basic PLANS functional requirements and to establish the structure from which higher level capabilities could be specified in the form of library routines (modules). Although specification of the library routines associated with both the Operations Model and the algorithms was to be a primary activity in the second part of Phase 1, a preliminary list of library modules did result from the analysis activities. In fact, the preliminary list was modified substantially during the second part of the Phase 1 study. However, it provided a useful starting point from which a careful examination of appropriate library contents could proceed.

3.9 PRELIMINARY ASSESSMENT OF LANGUAGE TRANSLATION OPTIONS

Investigation of PLANS implementation options began during the first part of the study. At issue was the basic mechanism for converting PLANS programs to executable code. If a general-purpose programming language existed with sufficient power to perform the basic operations implied by the basic functions of PLANS, it appeared desirable to implement PLANS by constructing a translator that uses the general-purpose language as its object

language. This approach is much less expensive than building a compiler to translate PLANS directly to machine language because the object language's compiler fills this function. Furthermore, this approach provides considerable machine-independence and allows use of additional high-level operations that may already exist in the object language at very little cost.

The first activity under this milestone was analysis of existing general-purpose programming languages to find those appropriate for use as object languages. A summary of the conclusions is shown in Table A-6. Translation to PL/I offers some major advantages. First, the implementation of the PLANS language translator is substantially less expensive and would require less time than building a complete new compiler or translating to another language. Second, it should be possible to build such a translator so that PL/I statements are admissible in the same programming as PLANS statements. Thus, the entire capability of PL/I would be available to the programmer even though he is not required to understand PL/I. Third, the translation to an existing language initially does not preclude implementation of a PLANS compiler (i.e., translator directly to machine language) at a later time. Thus, the decision was made to recommend as initial implementation mode, translation from PLANS to PL/I.

In the first six-month period of the study Martin Marietta subcontracted with Dr. James VanDoren of the Department of Computing and Information Sciences, Oklahoma State University, to deliver a two-day seminar on syntax-directed compilation and

and methods of implementing scheduling languages. This seminar, conducted in the seventh month (January 1974) of the study, proved highly useful and led to a formal method of semantic specification described in the next section.

Table A-6
Evaluation of Relevant Capabilities of Candidate Object Languages

Language / Feature	Arithmetic	List Processing	Indirect Data Reference	Indirect Subroutine Reference	Available with 360 OS	I/O Format Control	String Manipulation	Array Data	Well-Known	Commercial Support	Future Available
PL/I	+	+	+	+	+	+	+	+	+	+	+
FORTRAN	+	-	-	-	+	+	-	+	+	+	+
SIMSCRIPT	+	0	-	-	+	+	0	+	+	+	+
ALGOL	+	-	-	-	+	+	-	+	+	+	+
COBOL	+	-	-	+	+	+	-	+	+	+	+
SNOBOL	+	-	+	-	+	+	+	+	-	-	+
APL	+	+	0	-	+	-	+	+	-	+	+
LISP	0	+	+	-	-	-	-	-	-	-	-
IPL-V	-	+	+	+	-	-	-	-	-	-	-
GPSS	Specialized language. Clearly not appropriate.										

Legend: Function can be Performed:

+	Easily
0	Clumsily
-	For Practical Purposes, Not At All

4.0 MAJOR STUDY ACTIVITIES IN SECOND PART OF STUDY PHASE 1

4.1 DEVELOPMENT OF MECHANISM FOR SYNTACTIC/SEMANTIC SPECIFICATION

To enable a digital computer to "understand" and execute the commands of a programming language, a compiler or translator (computer program) must be developed to read the language statements and cause machine operations to occur in accordance with these commands. This requires two kinds of basic information: (1) a description of the allowable combination of language elements in a statement and (2) what must happen or what the computer must do after each language element or combination is recognized and accepted for execution. Therefore, the problem is to specify a language in this framework so the compiler or translator needed to implement the language can be programmed with minimal difficulty, ambiguity, and uncertainty of results.

Language syntax defines the rules by which language elements can be combined legally to form language statements. Most programming language syntaxes can be concisely defined with formal notational techniques such as the often used Backus-Naur Form (BNF).* BNF is a formal metalanguage for phrase-structure grammars whose application is not limited to any particular language. Thus, the scheduling language syntax could be concisely and unambiguously specified with existing techniques.

---

*Naur, P. (Ed): *Report on the Algorithmic Language ALGOL 60.* Communications of the ACM. 1960, 3, 299-314.

However, the specification of scheduling language *semantics*, i.e., the meaning of the language elements and statements, presented a different problem. A metalanguage is a language used to talk about a language. Natural languages, such as English, are in fact metalanguages when used to talk about a language. Semantic specifications for programming languages have frequently taken the form of written English text that describes what is supposed to happen when language statements are executed. Using English as a metalanguage it is:

1) difficult to be complete, i.e., to describe the results of all possible legal statements in the language;
2) difficult, in most cases, to be precise and unambiguous;
3) impossible to be concise enough to communicate to the language implementer effectively;
4) difficult to assure internal logical consistency in the language semantics;
5) difficult to provide insight on how various capabilities could be implemented in the compiler or translator.

A technique was sought to avoid these problems and to make the PLANS semantic specifications as precise as the syntactic specifications. The idea of embedding the semantics into the syntactic specification was suggested by Dr. James VanDoren during his January 1974 seminar. The embedding was accomplished by defining an elementary conceptual device, called a pseudomachine, which could respond to simple commands. The semantics of PLANS statements could then be defined in terms of these simple commands

that can be generated by the translator as the syntax of the statement is recognized. Thus, the pseudomachine commands, which contain the meaning of a PLANS statement, have a correspondence to the syntax or grammatical structure of that statement, and once the syntax of the statement is recognized, the semantics of that statement is known unambiguously.

The pseudomachine chosen for use in the PLANS specification mechanism involved a simple device whose basic data structure is a push-down stack. The data elements in such a stack may be addresses in computer storage, character strings, or logical values (true or false). An example description of the pseudomachine operations used to support the PLANS embedded semantic specifications is given in Table A-7, which shows that each of the pseudomachine operations is identified with a symbolic label, e.g., DUP, POP, INVERT, etc. A functional description of the operations is also given in English text, followed by a stack manipulation/transformation example where applicable. Thus, in the table, the operation DUP or DUPLICATE means "Push a copy of the content of Position 1 onto the stack." If the stack initially contained two data elements, XXXXXX and YYYYYY, which were considered to be in these relative positions,

XXXXXX

YYYYYY

then XXXXXX occupies Position 1 on the stack and the result of executing the commanded operation DUP would be to transform the stack to

Table A-7 PLANS Pseudomachine Operation List (Examples)

STACK OPERATIONS (EXAMPLES)

```
DUP                                    (DUPLICATE)
          PUSH A COPY OF THE CONTENT OF POSITION 1 ONTO THE STACK.
          E.G.,   DUP   RESULTS IN THE TRANSFORMATION:
                    XXXXXX              XXXXXX
                    YYYYYY     -->      XXXXXX
                                        YYYYYY

POP                                    (POP)
          POP THE CONTENT OF POSITION 1 OFF THE STACK.
          E.G.,   POP   RESULTS IN THE TRANSFORMATION:
                    XXXXXX     -->      YYYYYY
                    YYYYYY
```

ARITHMETIC OPERATIONS (EXAMPLES)

```
ADD                                    (ADD)
          ADD THE CONTENTS OF POSITIONS 1 AND 2, REPLACE THE CONTENT
          OF POSITION 2 BY THE RESULT, POP 1 POSITION.
          E.G.,   ADD   RESULTS IN THE TRANSFORMATION:
                    23                  29
                    6          -->      XXXXXX
                    XXXXXX              YYYYYY
                    YYYYYY

SUB                                    (SUBTRACT)
          SUBTRACT THE CONTENT OF POSITION 1 FROM THAT OF POSITION 2,
          REPLACE THE CONTENT OF POSITION 2 BY THE RESULT, POP 1 POSITION.
          E.G.,   SUB   RESULTS IN THE TRANSFORMATION:
                    23                  -17
                    6          -->      XXXXXX
                    XXXXXX              YYYYYY
                    YYYYYY

MULT                                   (MULTIPLY)
          MULTIPLY THE CONTENTS OF POSITIONS 1 AND 2, REPLACE THE CONTENT
          OF POSITION 2 BY THE RESULT, POP 1 POSITION.
          E.G.,   MULT   RESULTS IN THE TRANSFORMATION:
                    12                  36
                    3          -->      XXXXXX
                    XXXXXX              YYYYYY
                    YYYYYY

DIV                                    (DIVIDE)
          DIVIDE THE CONTENT OF POSITION 1 INTO THE CONTENT OF POSITION 2,
          REPLACE THE CONTENT OF POSITION 2 BY THE RESULT, POP 1 POSITION.
          E.G.,   DIV   RESULTS IN THE TRANSFORMATION:
                    12                  .25
                    3          -->      XXXXXX
                    XXXXXX              YYYYYY
                    YYYYYY
```

XXXXXX

XXXXXX

YYYYYY

All pseudomachine operations described in Table A-7 can be interpreted in a similar manner.

After the pseudomachine commands appropriate for defining the semantics of a given language have been specified, a basis exists for understanding the meaning of the language elements in a language statement. An example of the PLANS specifications using the pseudomachine as a mechanism for embedding the semantic specification in a BNF-type grammar is shown in Table A-8. The complete PLANS specifications in this format are in Volume III of this report. Previous use of this technique for functionally specifying a language is not known. Semantic definition by means of an abstract machine was incorporated in a cumbersome way in the Vienna Definition Language, but has been an otherwise undeveloped method.

A significant extension of the pseudomachine functional specification provided a means of translating language statements into executable code before actual implementation of a formal PLANS translator. To do this, the push-down stack device was modeled as a software machine (an emulator) by a computer program. This computer program simulated operations of the pseudomachine, which by design described what the computer was to do to execute each basic language operation. By using the PLANS grammar with the

Table A-8
PLANS Grammar with Embedded Semantics: a Format Illustration

```
            /*********************************************/
            /* INSERT, GRAFT, AND GRAFT INSERT STATEMENTS */
            /*********************************************/

INSERT_STATEMENT  :=
     "INSERT"
     .SET(TREE_STRING_ARITH_SWITCH = 1)
     CONSTRAINED_EXPRESSION
     ( "BEFORE" | "AS" )
     HARD_TREE_NODE
     .OUT(INSERT/INVERT)
     ( .TEST(REAL_DUMMY_SWITCH = 2)  .OUT(SNIP/INVERT)
       ( .TEST(LABEL_SUBSCRIPT_SWITCH = 1)  .OUT(G_ND)
       | .EMPTY  .OUT(GWL_ND) )
     | ( .TEST(LABEL_SUBSCRIPT_SWITCH = 1)  .OUT(R_ND)
       | .EMPTY  .OUT(RWL_ND) ) ) ;

GRAFT_STATEMENT  :=
     "GRAFT"
     .SET(TREE_STRING_ARITH_SWITCH = 1)
     ( "INSERT"
       CONSTRAINED_EXPRESSION
       .OUT(SNIP)
       ( "BEFORE" | "AS" )
       HARD_TREE_NODE
       .OUT(INSERT/INVERT)
     | CONSTRAINED_EXPRESSION
       .OUT(SNIP)
       "AT"
       HARD_TREE_NODE )
     ( .TEST(LABEL_SUBSCRIPT_SWITCH = 1)
       .OUT(G_ND)
     | .EMPTY
       .OUT(GWL_ND) ) ;

   /*************************************************************************/
   /* THESE THREE STATEMENTS COMPLEMENT THE TREE ASSIGNMENT                 */
   /* STATEMENT.  THE FOUR STATEMENTS ALLOW THE PROGRAMMER TO EITHER        */
   /* REPLACE AN EXISTING NODE OR INSERT AT ITS POSITION, MOVING IT         */
   /* AND ALL LATER NODES TO THE RIGHT, AND THEY ALLOW THE                  */
   /* PROGRAMMER TO EITHER PRUNE THE SOURCE STRUCTURE OR LEAVE IT           */
   /* UNALTERED.                                                            */
   /*                                                                       */
   /* IT SHOULD BE NOTED THAT THE TREE ASSIGNMENT STATEMENT IS BASIC        */
   /* AND IS SUFFICIENT TO PROVIDE ALL THE NECESSARY FUNCTIONAL             */
   /* CAPABILITIES.  FOR EXAMPLE,                                           */
   /*        GRAFT $A(1) AT $B(2) ;                                         */
   /* COULD BE ACCOMPLISHED BY                                              */
   /*        $B(2) = $A(1) ;                                                */
   /*        PRUNE $A(1) ;                                                  */
   /* IT SHOULD BE UNDERSTOOD, HOWEVER, THAT THE GRAFT FUNCTION             */
   /* ACCOMPLISHES THE OPERATION MUCH MORE EFFICIENTLY, SINCE IT             */
   /* NEED ONLY "MOVE" THE STRUCTURE BY CHANGING SOME POINTERS,             */
   /* SAVING A COMPLETE COPY OPERATION AND A COMPLETE PRUNE                 */
   /* OPERATION.  WHEN INSERT, GRAFT, AND GRAFT INSERT ACCOMPLISH           */
   /* THE DESIRED FUNCTION, THEY SOULD BE USED IN PREFERENCE TO             */
   /* FUNCTIONALLY EQUIVALENT PROGRAMMER-GENERATED CODE.                    */
   /*************************************************************************/
```

Note: Detailed Notational Definitions are found in Volume III of this report.

embedded semantics, each language statement was manually translated into appropriate pseudomachine operations; after input to the emulator, it was possible to use the emulator to perform all the machine operations necessary to execute the PLANS statements.

It should be emphasized that the software pseudomachine was not necessarily an efficient device for program execution; however, it provided a capability to test logical PLANS code and assure the adequacy of the language functional capabilities. More significant is the fact that executability was provided during the language definition phase rather than after implementation effort had occurred. It is important to note that the emulator mechanization of the pseudomachine provided a self-checking verification of the consistency and logical sufficiency of the PLANS functional specifications themselves much earlier than previous methods would have permitted.

4.2 EVALUATION OF LANGUAGE SUITABILITY FOR APPLICATIONS

In the first six months of the study, trial programming was done using a temporary syntax for a language with the functional capabilities anticipated for PLANS. This work is referred to in Section 3.3 as synthetic programming. After developing a syntactic and semantic specification mechanism, and after concluding to recommend translation from PLANS to PL/I, the syntax of PLANS developed rapidly. Because the conventions of PLANS coding needed to be similar to those of PL/I, a PL/I type syntax was adopted. Coding in PLANS rather than a functionally similar synthetic language (called TRIAL PLANS in some documentation) could then be accomplished.

Several applications programs or program segments were written to validate the adequacy of both the syntax and semantics of PLANS. As a result of these exercises, several new features appeared in the language that made manipulations of the data structures (i.e., trees) easier to perform. For example, GRAFT and PRUNE took on language meanings similar to their physical meanings. Alternatives were resolved about whether a label is preserved when its corresponding node is grafted onto another tree, etc. To illustrate the increase in statement power that resulted after the basic language capabilities were defined and the syntax and semantics were nearly developed, Table A-9 is presented. Table A-9 compares the TRIAL PLANS code developed early in the study with the PLANS code developed later. It should be stressed that the functions of the routines are identical.

Because of the availability of the pseudomachine described in Section 4.1, analysis of language suitability could include the execution of PLANS code. Several modules were coded in PLANS and the PLANS code manually converted to pseudomachine instructions using the PLANS grammar (i.e., the PLANS grammar with the embedded semantics). The pseudomachine instructions were then input to the computer program that emulated the pseudomachine, and the PLANS code logic was executed. This process served to verify (1) the adequacy and consistency of the specifications for the PLANS statements, (2) the validity of the pseudomachine emulator program, and (3) the adequacy and consistency of the logic of the module written in PLANS.

Table A-9 Development of Syntax and Semantics of PLANS

```
     This routine takes an unordered list of Jobs and orders it so that
     no precedence relationships are violated.

     Written in Trial PLANS:

     SUBROUTINE ORDERBYPREDECESSORS ( $SET , $REMAINDER )
     DO 2  I = 1, NUMBER ( $SET )
2    IF ( PREDECESSORS OF $SET(1) .SUBSET OF. $NAMES ) GO TO 3
     $REMAINDER = $SET
     $SET = $TEMP
     RETURN
3    $TEMP = $TEMP & $SET(I)
     $NAMES = $NAMES & NAME OF $SET(1)
     $SET = $SET   $SET(I)
     IF ( $SET .NE. $NULL )  GO TO 1
     $SET = $TEMP
     $REMAINDER = $NULL
     RETURN END

     Written in PLANS:

     ORDER_BY_PREDECESSORS:  PROCEDURE ($JOBLIST, $ORDERED_LIST) ;
          DECLARE $NAME_LIST, $TEMP LOCAL ;LOOP:
          GRAFT $JOBLIST.FIRST: (ELEMENT.PREDECESSOR SUBSET OF $NAME_LIST)
               AT $TEMP ;
          IF $TEMP IDENTICAL TO $NULL THEN RETURN ;
          $NAME_LIST(NEXT) = LABEL ($TEMP) ;
          GRAFT $TEMP AT $ORDERED_LIST(NEXT) ;
          GO TO LOOP ;
          END ORDER_BY_PREDECESSORS ;
```

An example of executed PLANS code is contained in Table A-10 which shows the input data and output data for the ORDER_BY_ PREDECESSORS code of Table A-9. The data pertain to a simple network representation of Shuttle Operations also shown in Figure A-8.

Several examples of coded routines or programs are included in Volume II of this report. All examples were developed during the study as a means of evaluating the applicability of the language.

Table A-10 Execution of ORDER_BY_PRECESSORS *Coded in PLANS*

INPUT_DATA

```
$JOBLIST
     LAUNCH_OPS
          PREDECESSORS
               X  -  SERVICE_SHUTTLE
               X  -  PREP_CREW
     ONORBIT_OPS
          PREDECESSORS
               X  -  LAUNCH_OPS
     DEORBIT_LAND
          PREDECESSORS
               X  -  ONORBIT_OPS
     CREW_TRN_OPS
          PREDECESSORS
               X  -  PAYLOAD_OPS
     PREP_CREW
          PREDECESSORS
               X  -  MISSION_BRIEF
     PAYLOAD_OPS  -
     PREP_DROP_TANK  -
     SERVICE_SHUTTLE
          PREDECESSORS
               X  -  MATE_ORBITER
     DEBRIEF_CREW
          PREDECESSORS
               X  -  DEORBIT_LAND
     MISSION_BRIEF
          PREDECESSORS
               X  -  CREW_TRN_OPS
     PREP_ORB_LAUNCH
          PREDECESSORS
               X  -  PAYLOAD_OPS
     ASSEMBLE_SRBS  -
     MATE_TANK_TO_SRB
          PREDECESSORS
               X  -  ASSEMBLE_SRBS
               X  -  PREP_DROP_TANK
     MATE_ORBITER
          PREDECESSORS
               X  -  MATE_TANK_TO_SRB
               X  -  PREP_ORB_LAUNCH
     REFURB_PAD
          PREDECESSORS
               X  -  LAUNCH_OPS
     REFURB_LUT
          PREDECESSORS
               X  -  LAUNCH_OPS
     RECYCLE_SRB
          PREDECESSORS
               X  -  LAUNCH_OPS
     RECYCLE_ORB
          PREDECESSORS
               X  -  DEORBIT_LAND
```

OUTPUT_DATA

```
$ORDERED_LIST
     PAYLOAD_OPS
     CREW_TRN_OPS
          PREDECESSORS
               X  -  PAYLOAD_OPS
     PREP_DROP_TANK  -
     MISSION_BRIEF
          PREDECESSORS
               X  -  CREW_TRN_OPS
     PREP_CREW
          PREDECESSORS
               X  -  MISSION_BRIEF
     PREP_ORB_LAUNCH
          PREDECESSORS
               X  -  PAYLOAD_OPS
     ASSEMBLE_SRBS  -
     MATE_TANK_TO_SRB
          PREDECESSORS
               X  -  ASSEMBLE_SRBS
               X  -  PREP_DROP_TANK
     MATE_ORBITER
          PREDECESSORS
               X  -  MATE_TANK_TO_SRB
               X  -  PREP_ORB_LAUNCH
     SERVICE_SHUTTLE
          PREDECESSORS
               X  -  MATE_ORBITER
     LAUNCH_OPS
          PREDECESSORS
               X  -  SERVICE_SHUTTLE
               X  -  PREP_CREW
     ONORBIT_OPS
          PREDECESSORS
               X  -  LAUNCH_OPS
     DEORBIT_LAND
          PREDECESSORS
               X  -  ONORBIT_OPS
     DEBRIEF_CREW
          PREDECESSORS
               X  -  DEORBIT_LAND
     REFURB_PAD
          PREDECESSORS
               X  -  LAUNCH_OPS
     REFURB_LUT
          PREDECESSORS
               X  -  LAUNCH_OPS
     RECYCLE_SRB
          PREDECESSORS
               X  -  LAUNCH_OPS
     RECYCLE_ORB
          PREDECESSORS
               X  -  DEORBIT_LAND
```

Note: **The symbol X is used in these structures for a null label**

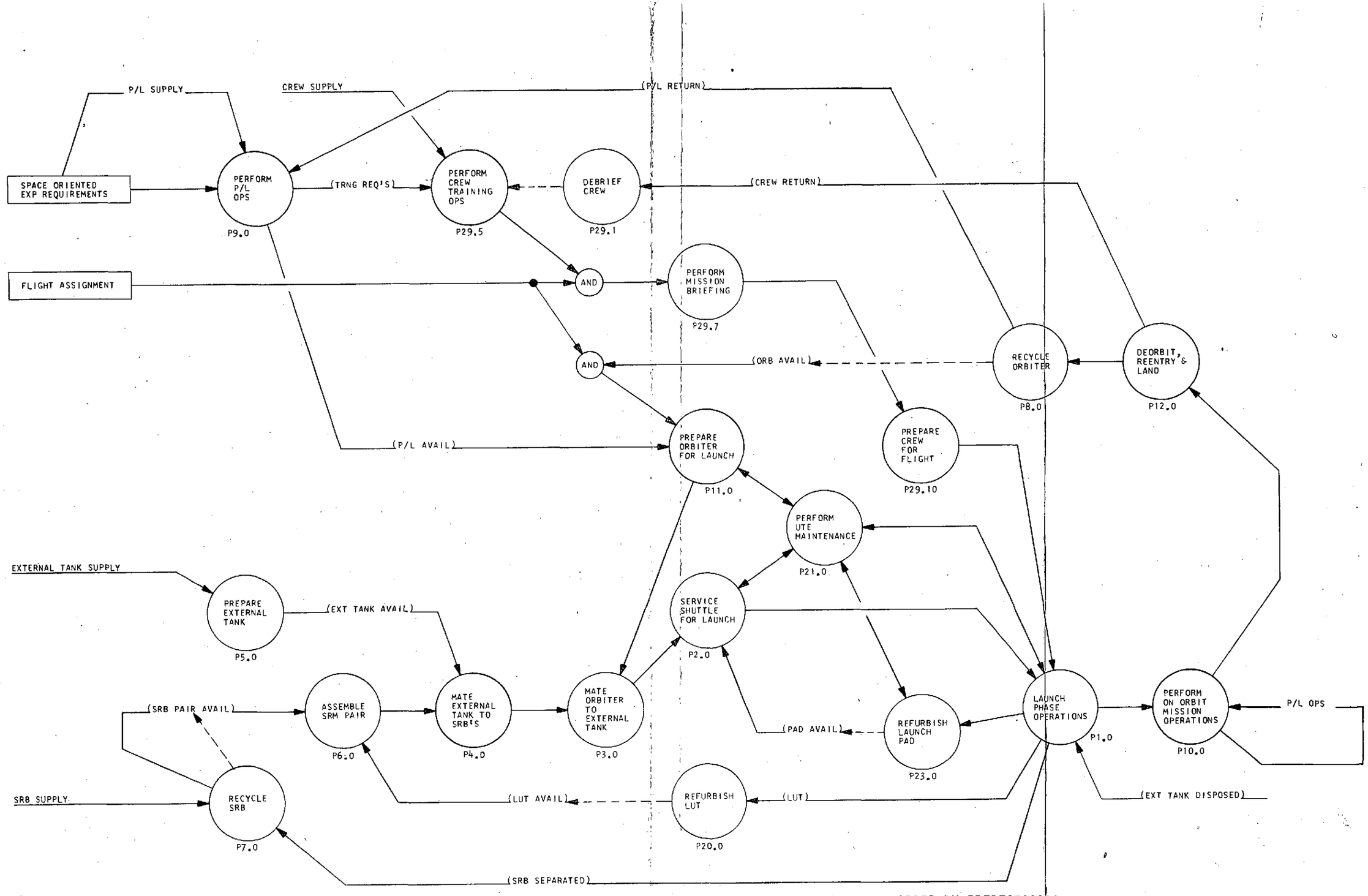


Fig. A-8 ORDER_BY_PREDECESSORS: *Example Problem for Shuttle Top Flow*

FOLDOUT FRAME 1

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

FOLDOUT FRAME 2

4.3 EVALUATION OF LANGUAGE IMPLEMENTATION FEASIBILITY

Having arrived at a desirable basic functional design for PLANS, it was necessary to consider the feasibility of its implementation. This required a consideration of specific execution mechanisms for individual PLANS statements, language parsing and translator design mechanisms, system implications of PLANS, and possible disc access and update mechanisms. In each case, the emphasis was not on making detailed tradeoff decisions, but on a determination that at least one feasible method existed.

Dynamic tree manipulation is the basis of PLANS. The most basic implementation feasibility issue is, therefore, the determination of a mechanism for the representation of dynamic trees. The mechanism that has been selected is the binary tree structure. In this structure, a nonterminal node contains a pointer to its leftmost descendant, and each nonrightmost node at a given level points to its next sibling to the right. This structure is simple to implement and is quite efficient for most foreseen applications, but random access time by subscript or label varies in porportion to the number of nodes per level. Any mechanism that avoids this problem would necessarily require multiple descendant pointers, with increased overhead, and hashed or ordered label pointers to allow a nonsequential search. These methods would seriously degrade performance with small trees and have been rejected as difficult and undesirable.

Storage of labels and values is on issue that might well decide the efficiency of PLANS execution. It is a very simple matter to physically store this information as part of each node, but this requires allocation of label and value space for each node, even though a given node may lack one or both. It also requires allocation of full-length fields (or variable-length node records) for this information, even though the actual information to be stored requires only a small portion of this space. This situation is amenable to the usual space-time tradeoff, and we have elected to employ a variant of the buddy system to allow dynamic allocation of varying-length records for label and value storage. Some testing of this method will be required before it can be determined whether it represents a reasonable tradeoff, but preliminary execution trials indicate its basic feasibility and practicality.

A second concern was with mechanisms appropriate for language parsing and translation. The PLANS syntax proves to be expressible in a form that is amenable to top-down deterministic parsing, a simple and efficient technique. Furthermore, the PLANS functional specification was expressed in a form quite amenable to the application of automated translator-writing concepts. These methods have now been employed to generate a fairly extensive syntax checker for PLANS, and a very rudimentary code generator. The indicated parsing method and the automated translator generation approach appear quite powerful and appropriate for this application.

PLANS has, or could have if extended in foreseeable ways, several implications at the computer system level. The most basic of these derives from the extreme desirability of PL/I as the translator object language. If PLANS is to be executed on a particular system, translator development and operation will be much more straightforward if that system has a PL/I capability. In the past, this would have restricted PLANS to IBM computers, but this is clearly no longer true. CDC and Univac have announced delivery of PL/I compilers in 1975, and other major manufacturers are quite likely to develop compilers for it.

The possible extension of PLANS into interactive programming, interactive execution, and disc access/update, particularly using a generalized data base management system, has obvious system implications, but these implications are not significantly PLANS-specific. All the usual considerations that are encountered in the development of interactive systems and data base applications can be expected with these extensions.

The use of a generalized data base system, warranted special consideration, since PLANS tree structures represent a well-defined special application for such a system. System 2000 was considered since it is a simple, easy-to-use, hierarchical data base management system. This system offers a subset of the functional capabilities of most other such systems, but makes the capabilities very accessible to the user. Analysis revealed good compatibility between PLANS and System 2000, and it was concluded that an automatic translation capability to map PLANS statements

into System 2000 statements is feasible. Since System 2000's capabilities also exist in such systems as IBM's IMS, it is apparent that PLANS access and update statements can be made to functionally correspond to operations in those systems, but perhaps at some cost in difficulty and complexity.

4.4 DEVELOPMENT OF CONTENTS AND SPECIFICATIONS FOR LIBRARY MODULES

The major emphasis of the modeling and algorithm tasks in the second part of Phase 1 was placed on definition of the contents of a module library and determination of the correct separation of functions among the modules. A detailed analysis of scheduling problems reveals a very large number of capabilities that could be preprogrammed; yet many of these are useful only in highly specialized problems and thus would have little value in a general library. The problem with specifying a program library is not so much what to put in it, as what to leave out of it.

During the second part of Phase 1, modules that met the following criteria were specified:

1) Each module is limited to a single logical function. Although it is possible to group several of the specified modules together based on high-level functional similarity, to do so would restrict flexibility or decrease the computational efficiency of the functions represented. Therefore, the modules specified for the program library should perform a single, separable logical function.

2) Each module performs a function that is common or likely to occur in typical scheduling software. No module is specified that is applicable only to an infrequent special case, one that is required only in an unenlightened or highly encumbered approach where an alternative exists.

3) No module specified contained judgments or decision making logic for which the criteria are open to opinion. For example, no module should assume a specific economic model, a queuing service policy, or a criterion for resolving resource alternatives. These judgmental matters are considered too problem-dependent and inflexible for an initial library specification. Because of the criterion for functional simplicity and separability (criterion 1), the specified modules perform elementary operations and generally return information upon which decisions can be made rather than making the decisions themselves. Modules that make simple decisions based on quantitative criteria, which are easily perceived by the user, are specified as decision algorithms. A clear distinction is preserved between simple decision making modules (algorithms) and information providing modules (the operations model). Thus, all of the latter are equally applicable whether exercised interactively by a user making real-time decisions or in a batched system design where algorithm modules make the scheduling decisions.

The output of this analysis was specifications for modules covering a range of sophistication, from computing the duration of an interval to calculating the entire schedule for a project with tens of thousands of jobs each sharing resources with other jobs. The contents of the specified module library are shown in Table A-11 classified loosely according to their functions within the overall modeling and solution process.

Some generalizations are evident from an examination of the library contents. Two major types of solution strategies are supported: mathematical programming techniques and project scheduling techniques. Of the two, it was concluded that the project scheduling techniques represent the most capable and practical generalized techniques available for realistic problems. Mathematical programming techniques are useful, however, for special problems with small dimensionality, and are, therefore, supported by the specification of appropriate library modules.

It is also evident that the library contains many modules that perform the common bookkeeping functions that can be standardized without loss of logic flexibility. For example, all scheduling programs must keep track of resource assignments as they are made. This simple function is accomplished by the modules UPDATE_RESOURCE and WRITE_ASSIGNMENT. Similarly, all scheduling involves the checking of real or anticipated assignments for constraint compatibility. Four modules that perform constraint checking are specified. To facilitate the formulation of logically consistent operations model definitions, three preprocessing modules are

Table A-11 Contents of the Module Library by Title

PREPROCESSORS

- CHECK_FOR_PROCESS_DEFINITION
- NETWORK_EDITOR
- REDUNDANT_PREDECESSOR_CHECKER

PRELIMINARY PROCESSORS

- GENERATE_JOBSET
- PREDECESSOR_SET_INVERTER
- ORDER_BY_PREDECESSOR
- CRITICAL_PATH_PROCESSOR
- PREDECESSOR_SET_INVERTER
- NETWORK_ASSEMBLER
- CRITICAL_PATH_CALCULATOR
- CONDENSED_NETWORK_MERGER
- NETWORK_CONDENSER
- PROJECT_DECOMPOSER
- COMPATIBILITY_SET_GENERATOR
- FEASIBLE_PARTITION_GENERATOR

ELEMENTARY FUNCTIONS

- DURATION
- ENVELOPE
- CHECK_ELEMENTARY_TEMP_RELATION
- WRITE_ASSIGNMENT
- INTERVAL_UNION
- INTERVAL_INTERSECTION

PERFORMANCE OR CONSTRAINT STATUS

- CHECK_EXTERNAL_TEMP_RELATIONS
- CHECK_INTERNAL_TEMP_RELATIONS
- RESOURCE_PROFILE
- POOLED_DESCRIPTOR_COMPATIBILITY
- CHECK_DESCRIPTOR_COMPATIBILITY

DATA UPDATING

- UPDATE_RESOURCE
- UNSCHEDULE
- DESCRIPTOR_UPDATE

ALGORITHMS

- FIND_MAXIMUM
- FIND_MINIMUM
- HEURISTIC_SCHEDULING_PROCESSOR
- RESOURCE_ALLOCATOR
- RESOURCE_LEVELER
- NEXT_SET
- PRIMAL_SIMPLEX
- DUAL_SIMPLEX
- GUB_LP
- INTEGER_PROGRAM
- MIXED_INTEGER_PROGRAM

provided. Although many additional routines could be specified to perform analyses of input data, it was decided that the problem analyst with even a minimum of experience would be unlikely to use such capabilities, i.e., he would be more likely to have sufficient understanding of his problem to avoid certain obvious logical inconsistencies. For example, it was deemed unnecessary to build a module to check if any defined process requires more resources than are determined to be in the problem model. The inconsistencies that are more likely to occur are, however, detectable by the specified preprocessing modules.

Finally, the library contains many typical ordering and partitioning functions call preliminary processors. These modules calculate parameters (such as slack in a network) or create sequence lists that are often used by a decision algorithm. These same data are equally useful in an interactive scheduling process in which human decisions are used. Thus, minimum executive logic or scheduling system design has been assumed in specifying the contents of the module library.

The detailed functional specification of each module in the library is contained in Volume III of this report, whereas insight in the actual use of the library for scheduling is provided in Volume II.

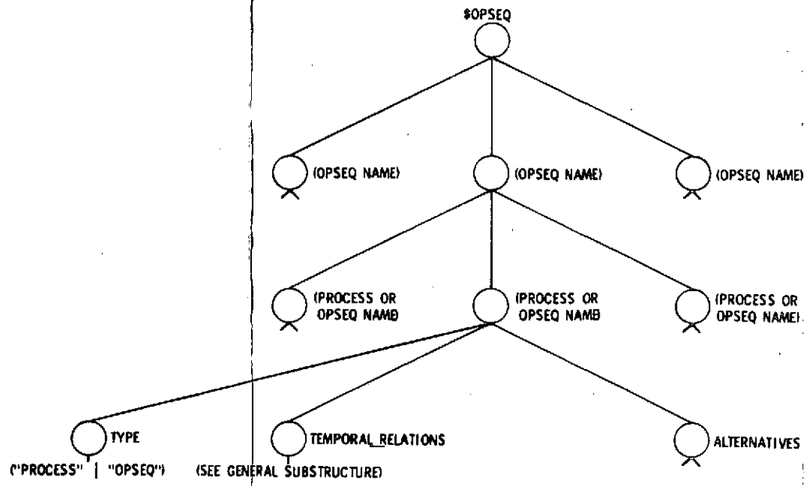
4.5 DEVELOPMENT OF STANDARD DATA STRUCTURES

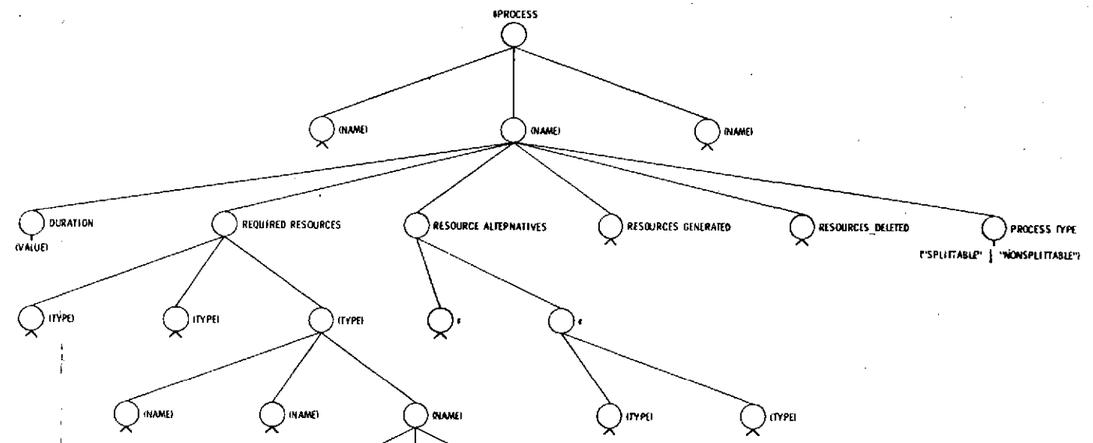
The utility of a module library depends not only on its contents and the appropriate allocation of functional capabilities, but also on the degree of integration between the modules. It is undesirable, for example, to require through improper specifications that the program designer devise elaborate special purpose tree structures and reformatting logic to convert the output of one module to the appropriate format for the input of another module. This problem is minimized if a set of generalized template data structures are defined. Use of these generalized structures can then be assumed by the library modules. These structures serve to integrate the modules and, in doing so, provide a framework within which the analyst can model the operational system to be scheduled.

A major activity in the second part of this study was to define the standard data structures in a manner that was nonrestrictive in terms of modeling flexibility. A prime consideration in structure definition was unambiguous interpretation of input and output information for a scheduling problem by problem analysts or by the logic of the problem library modules. It was discovered early in the study that basic system descriptive information was hierarchically related and that this same information separated rather clearly into three types of tree structures that we labeled $OPSEQ (a compression of operational sequence) $PROCESS, and $RESOURCE. In the second part of the study, the details of these structures were refined iteratively as the specifications for the module library became specific.

The final standard data structures are shown in Fig. A-9. To illustrate the evolution of these structures that transpired, consider the portion of the tree $RESOURCE with the label ASSIGNMENT. This substructure has been designed to record the results of executing scheduling decisions that assign the resources to jobs for particular intervals of time. The substructure design must accommodate the fact that the resource in question might be a single item or it might be a pool. Specifically, the resource might be CREWMAN_JONES or it might be the pool called ASTRONAUTS. A scheduling problem could require a mixture of pooled resources and item specific resources. Thus, a single standard structure for $RESOURCE must be designed only after a careful categorization of the possible model variations has been accomplished.

This particular study activity resulted in a set of definable characteristics for general problem models that must be considered in realistic scheduling problems. These definitions along with the standard data structures that accommodate the description of those characteristics are collectively called the general operations model. Table A-12 summarizes the results of this analysis. The terminology "explicit descriptors" is used to distinguish between resources whose descriptors do not change after being assigned and used in a job, and resources whose descriptors change as a result of being assigned and used in a job. An example of the latter is the descriptor LOCATION, which may be changed by scheduling and executing a job called DELIVER_PAYLOAD.





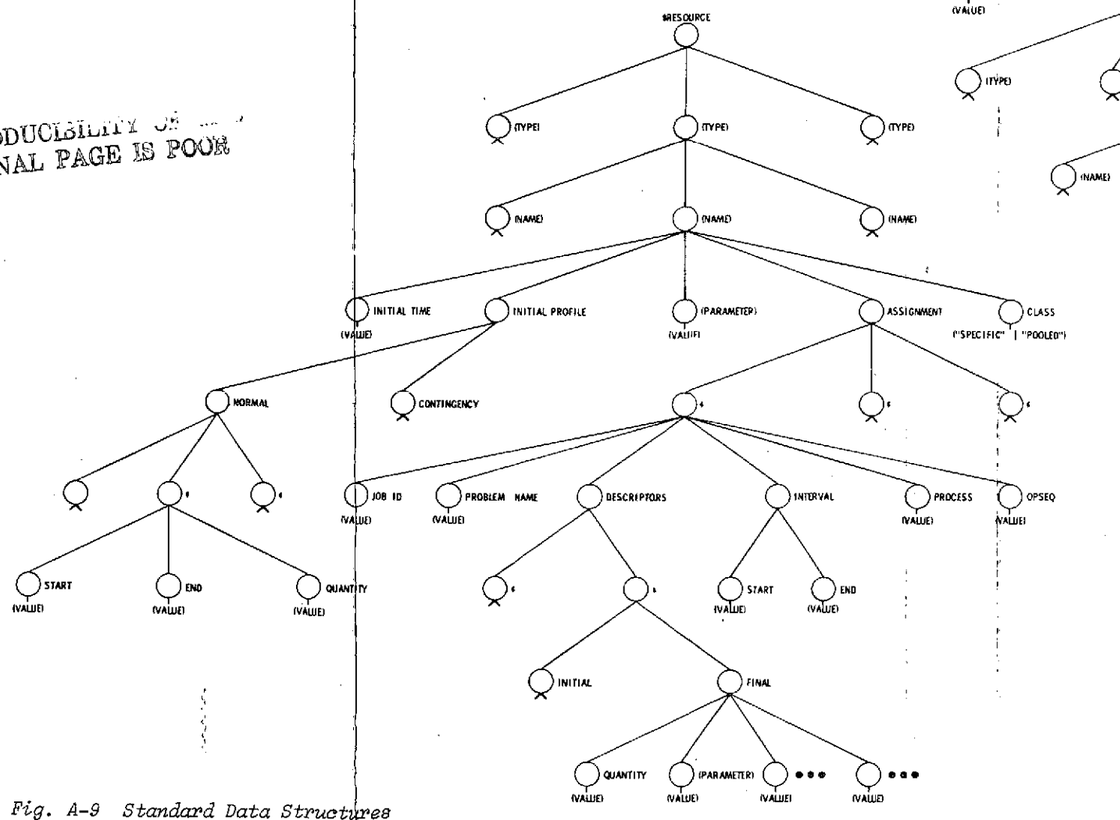


Fig. A-9 Standard Data Structures

Table A-12 Characterization of Problem Models

RELATIONS BETWEEN JOBS (TEMPORAL RELATIONS)

Simple Predecessors

Start of Job B $\geq$ End of Job A

Generalized Temporal Relations

$$\left\{\begin{matrix}\text{Start}\\\text{End}\end{matrix}\right\} \text{ of Job B } \left\{\begin{matrix}<\\\leq\\=\\>\\\geq\end{matrix}\right\} \left\{\begin{matrix}\text{Start}\\\text{End}\end{matrix}\right\} \text{ of Job A } \left\{\begin{matrix}+\\-\end{matrix}\right\} \text{ Constant}$$

RELATIONS BETWEEN JOBS AND THEIR REQUIRED RESOURCES

Job A Requires

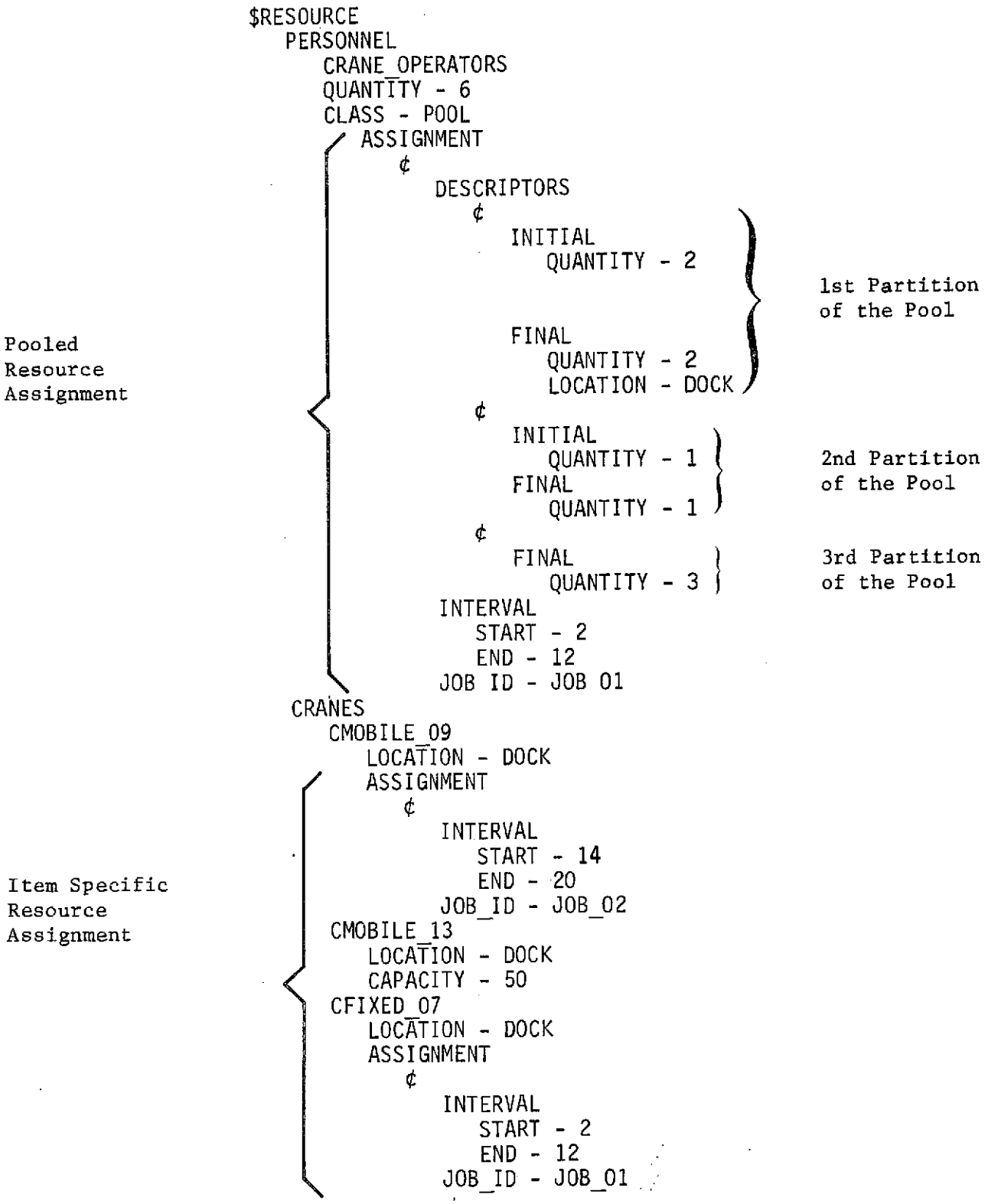
$$\left\{\begin{matrix}\text{One}\\\text{More Than One}\end{matrix}\right\} \left\{\begin{matrix}\text{Pooled}\\\text{Item-Specific}\end{matrix}\right\} \text{ Resource with } \left\{\begin{matrix}\text{No Explicit Descriptors}\\\text{Changeable Explicit Descriptors}\end{matrix}\right\}$$

or it requires any combination of the above for an interval of time that may or may not be the entire duration of the job.

The ASSIGNMENT substructure of $RESOURCE must be sufficiently flexible to handle the assignment information for any type of resource. Table A-13 shows an assignment structure for one pooled resource that has been partitioned several ways, each with a unique set of explicit descriptors and one item specific resource. It can be seen that both resource types are accommodated by the general structure shown in Fig. A-9.

Analyses were conducted that led to other similar general structures within $PROCESS and $OPSEQ. In addition, the output structures of the library modules were developed to have maximum compatibility with the three structures already discussed. The structure of $SCHEDULE shown in Fig. A-10 is an example of a specified standard structure that is obviously not an input to

Table A-13
The Assignment Substructure of $RESOURCE *for Pooled and Item-Specific Resources*



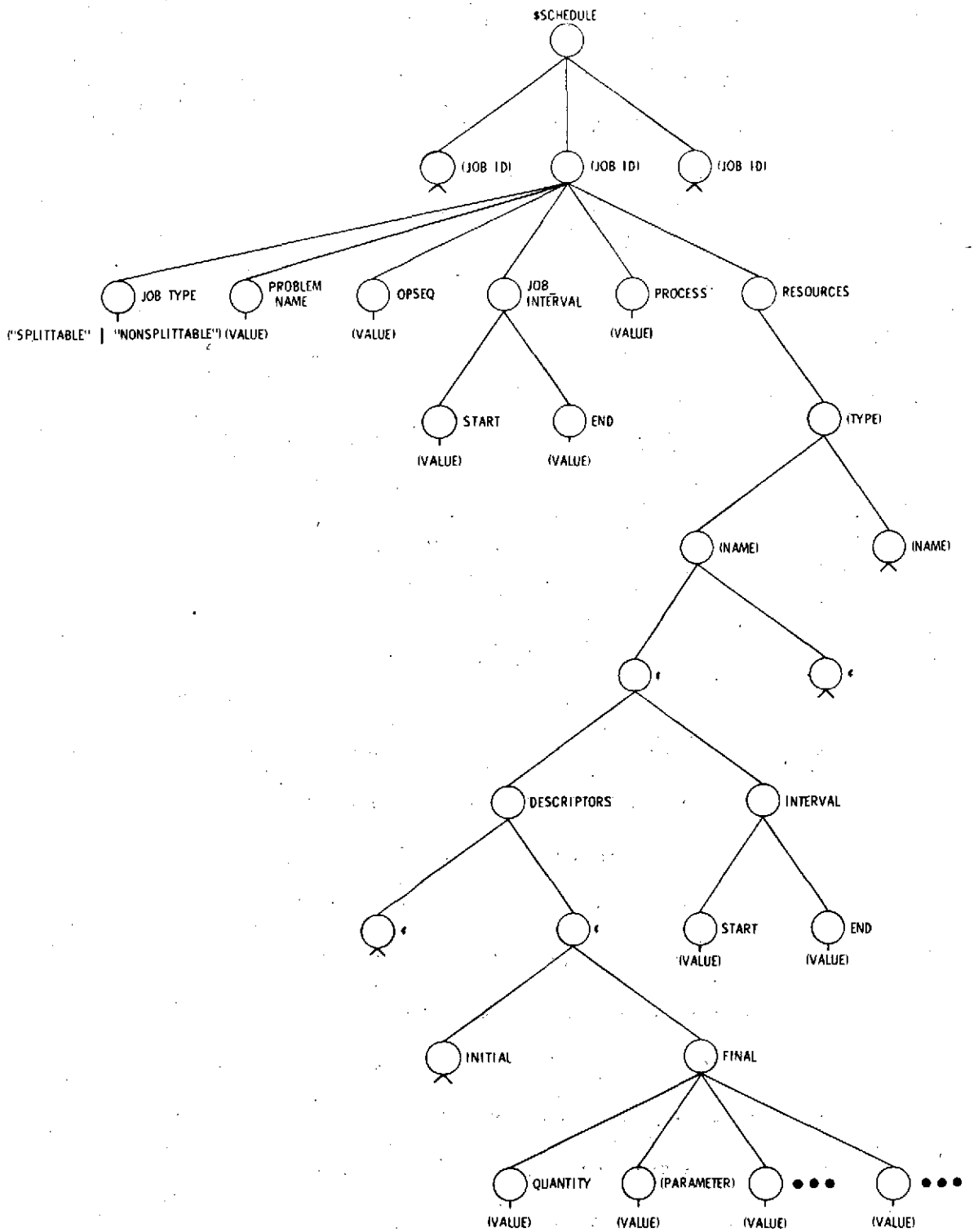


Fig. A-10 $SCHEDULE *Structure*

a scheduling program, but which should be standardized to facilitate the use of modules that check constraint violations or compute resource profiles. It can be noted that $SCHEDULE, $PROCESS and $RESOURCE have certain substructures that are identical. This is a result of recognizing that scheduling logic will consist of grafting or inserting portions of one tree on another, a procedure that is simple if the structures are common.

Other standard data structures are discussed in Volume II of this report. Volume II provides a complete description of how the structures accommodate the problem model variations identified by the analysis just described.

4.6 ASSESSMENT OF IMPLEMENTATION FEASIBILITY OF SPECIFIED MODULES

To provide data on the scope of the effort required to implement the modules being specified, selected modules were programmed. A range of functional characteristics was considered in selecting the modules to be coded. Simple bookkeeping-type functions that should be easily programmable in PLANS are represented by the modules shown in Group I of Table A-14. To verify the adequacy of the PLANS language, and the functional specifications as written, all the modules in Group I were coded in PLANS.

The modules shown in Group II of Table A-14 are typical of the more complex functions specified for the PLANS module library. Coding was generated for the modules of Group II in order to balance the implementation assessments gained while coding the low-level modules of Group I.

Table A-14 Modules Coded for Implementation Feasibility Analysis

GROUP I ELEMENTARY MODULES	
DURATION	Calculated the duration of any standard (simple or multiple) interval
ENVELOPE	Calculates an interval that is the smallest cover of a given standard (simple or multiple) interval
ORDER_BY_PREDECESSOR	Produces a list of jobs with the property that all jobs appear in the list only after all their predecessors have appeared; i.e., produces a nonunique technological ordering.
WRITE_ASSIGNMENT	Writes a single assignment for a resource and adds the assignment node in chronological order in $RESOURCE.
UPDATE_RESOURCE	Records the scheduling of a schedule unit (job) by writing assignments in $RESOURCE for all resources used in the schedule unit.
UNSCHEDULE	Deletes assignments from $RESOURCE for all resources associated with a specified job to be deleted.
RESOURCE_PROFILE	Determines the profile of a resource pool over a given time interval for both "normal" and "contingency" levels. Determines the profile of the assigned portion of a pool and gives the jobs to which the resources are assigned.
GROUP II HIGHER-LEVEL MODULES FROM OPERATIONS MODEL	
NEXTSET	Determines a set of specific resource items to meet the requirements of a job and permit the earliest possible execution of that job. Determines future times the job requirements can be met with any combination of appropriate resource types.
GENERATE_JOBSET	Creates individual jobs for each occurrence of a process specified explicitly or via an operations sequence in $OBJECTIVES. Merges information contained in $OBJECTIVES $OPSE and $PROCESS into a tree called $JOBSET. Jobs in $JOBSET are ready for the decision algorithms to make explicit assignments.
GROUP III ALGORITHM MODULES	
MIXED_INTEGER_PROGRAM	Solves linear programs that contain both continuous and integer-valued decision variables.
INTERGER_PROGRAM	Solves the linear form of the binary decision-making problem.
RESOURCE_ALLOCATOR	Allocates resources to jobs to satisfy all resource constraints and heuristically produce a minimum duration schedule.
RESOURCE_LEVELLER	Reallocates resources to smooth the usage of resources while maintaining schedule constraints.

Finally, several algorithm modules (Table A-14, Group III) were written to assess the effort required to implement sophisticated solution techniques. The results of these three coding analyses are summarized.

1) The adequacy of PLANS for implementing nonmathematical scheduling routines was verified.

2) The functional specifications for the modules coded were specific enough to provide a clear indication of what was needed, but not so detailed as to preclude options for detailed logic design. This conclusion was reached by using personnel to design and code modules in Groups I and II who had not previously been associated with study. Before starting the design and code exercise, they had no previous knowledge of PLANS or the operations model conventions.

3) Several extensions to the functional capabilities of the specified modules should be expected during implementation. It was found that careful logic design could provide output information that was additional to that specified without increasing the complexity or efficiency of the logic internal to the module. For example, the NEXTSET specification called for the return of the earliest availability window in which a resource set would be available to meet specified requirements. The logic needed to determine this earliest window also determined all other later windows in which the requirements could be met. This and other examples, which resulted from the implementation assessment task, lead to the conclusion that a

careful implementation effort should produce functional capabilities in excess of those specified.

4) Mathematical programming techniques can be expected to require greater implementation efforts than other relatively sophisticated scheduling modules. This results not only from the complexity of the mathematical logic but also from the need to use the most advanced mathematical programming methods to maximize the problem dimensionality that can be handled. Programmed in this study was a technique suggested by Geoffrion to adapt the well-known integer programming technique employing surrogate constraints to the problem decomposition derived by Benders. Computational implementation of this approach has not been reported elsewhere; detailed documentation of this program will appear subsequently. It is important to note here, however, that the development of state-of-the-art mathematical programming routines is sufficiently complex to suggest that a careful analysis of usage requirements be made before a general implementation effort is initiated.

5) PLANS provides appropriate capabilities to program project scheduling routines. Efforts required to implement such routines were less than anticipated.

To provide insight on how the various specified modules would integrate and to verify the adequacy of the standard data structures, a demonstration program was designed to solve a typical Shuttle flight scheduling problem. The architecture of the program is illustrated schematically in Fig. A-11. The implementation

of the demonstration problem was taken to a point where input data structures were defined and executive logic functionally specified. The analysis confirmed implementation feasibility for a program of this nature.

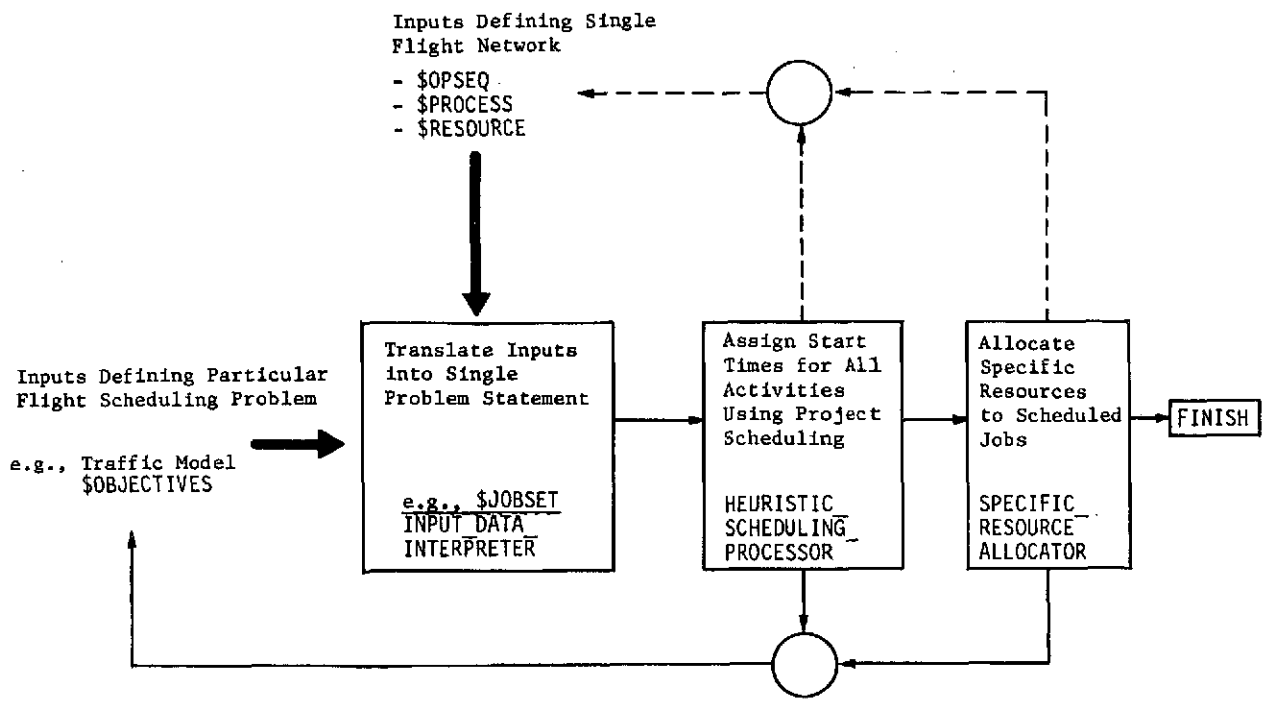


Fig. A-11 Demonstration Program Macrologic

It was discovered, however, that the $OPSEQ structure and the GENERATE_JOBSET module should be extended to incorporate information on "commonality" constraints. Commonality is a term that refers to coupling of resource allocation decisions across jobs. For example, if Job 32 and Job 33 each require an orbiter, and

the orbiter chosen must be the same orbiter, then a commonality constraint exists. This constraint appropriately belongs in the $OPSEQ standard data structure since it concerns information relating jobs to one another.

The formulation of a demonstration problem served to verify implementation of a functionally integrated program using PLANS routines and PLANS executive logic. It also served to extend capabilities of specified modules and data structures to make them cover more of the functions required in a typical problem.

4.7 ASSESSMENT OF METHODS FOR AUTOMATED ALGORITHM APPLICATION

The approach to identifying appropriate logic for module specification used in this study placed early emphasis on elementary and fundamental modules. An ultimate goal has been to progress upward in level, sequentially addressing more and more automated scheduling capabilities. An analysis on how realistically complex schedules are successfully generated leads to a single inescapable conclusion: *Human judgment is always present in the overall decision-making process if the resulting schedulings are realistic.* This fact suggests caution in proceeding toward greater automation.

The analysis performed in the task described here was limited to a consideration of how current project scheduling methodology, which can in fact handle realistic dimensionalities, can be used in solving problems with greater model generality than is directly accommodated by project scheduling models. A scenario of human/computer activities was developed to which subsequent analyses

can provide greater detail. The overall approach to heuristic scheduling using project scheduling methods consists of three major procedural elements:

1) Resource constrained project scheduling applied to limited problem descriptions;
2) Manual scheduling for fine tuning and resolving low-dimensional complex conflicts; and
3) Detailed resource tracing considering resource descriptors.

Table A-15 displays the expected frequency, problem size and interface needs for these three elements.

Table A-15 Strategy Characteristics

Strategy	Expected Frequency of Use	Problem Size	Interface
Project Scheduling	Very Often	$O(10^3)$	Batch
Interactive Perturbations	Very Often	$O(10)$	Interactive
Detailed Resource Tracing	Occasionally	$O(10^2)$	Batch

The project scheduling modules could easily be a mainstay of a scheduling system especially during the early phases of a new operation. Their forte is handling large problems of a simple format to give the scheduler a handle on an unknown situation. The quantities of data used and presented point to batch rather than interactive computer interfaces.

Interactive scheduling, employing user intuition and low level modules to schedule a small number of jobs is expected to occur at least as often as large project scheduling and probably more often as the operations become more routine or well-defined. In this

case a basic framework schedule exists and the man is resolving real-time conflicts or those which are difficult to express to the computer. Experience shows that once a man is familiar with the operation he is scheduling, he can resolve most conflicts *if he can see the effect of his decisions*. Interactive computer interfaces would greatly facilitate this process.

Detailed resource tracing may be necessary to ensure that all jobs needed to guarantee that proper resource states are in the network, or to validate a schedule. However, detailed resource tracing should be avoided most of the time for two reasons. First, the changing of states for a particular resource can usually be handled using jobs (e.g., job: receive payload rather than resource: payload, state: received.) Second, the generation of schedules that are too detailed ignores the fact that the future is never exactly what is expected. Such schedules limit the individual scheduler freedom to handle day-to-day crises and special allocations. The capability of the individual scheduler is a considerable resource in itself.

The analysis conducted under this task produced basic concepts that led directly to a preliminary concept for a man-computer scheduling system. This concept is illustrated in Fig. A-12. The utility of such a system depends heavily on the allocation of responsibilities between the computer and the human scheduler. Thus, a set of specific test objectives emerged; i.e., evaluate the performance of both the man and the computer system in the roles indicated in Fig. A-12. In particular, what functional elements

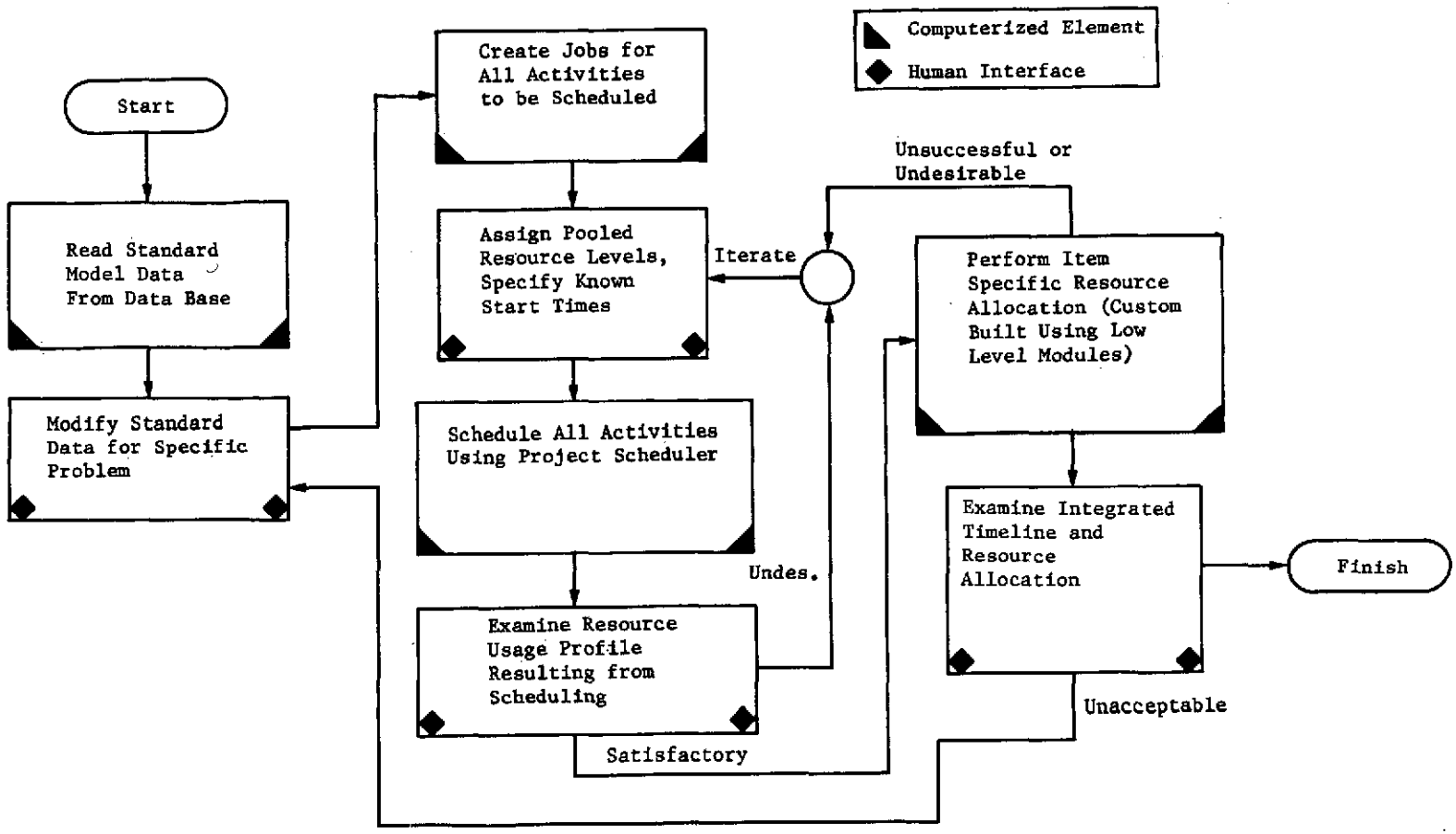


Fig. A-12 A Man-Computer Scheduling System Concept

belong in the iteration paths? It was decided to build the demonstration program with the interaction points indicated in Fig. A-12. Specific tests could then be run on proposed automated problem reformatting logic and on tutorial-type modules that might be placed in the feedback paths of the man-computer system concept. These tests will be executed in future analyses.